МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

С. О. Цибульник

Технології розробки програмного забезпечення-1

Комп'ютерний практикум

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології»,
освітньо-професійною програмою «Комп'ютерно-інтегровані технології та системи навігації і керування»*

Київ

КПІ ім. Ігоря Сікорського

2021

Рецензенти: *Гармаш О.В.,* канд. техн. наук, доцент кафедри акустичних та мультимедійних електронних систем КПІ ім. Ігоря Сікорського

Івасенко В.М., канд. техн. наук, асистент кафедри інформаційно-вимірювальних технологій КПІ ім. Ігоря Сікорського

Відповідальний редактор: *Павловський О.М.,* канд. техн. наук, доцент кафедри приладів і систем орієнтації і навігації КПІ ім. Ігоря Сікорського

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 7 від 13.05.2021 р.) за поданням Вченої ради Приладобудівного факультету (протокол № 3/21 від 29.03.2021 р.)

Електронне мережне навчальне видання

Цибульник Сергій Олексійович, канд. техн. наук, доцент

Технології розробки програмного забезпечення-1

Комп’ютерний практикум

Цибульник С. О. Технології розробки програмного забезпечення-1. Комп’ютерний практикум [Електронний ресурс] : навч. посіб. для студ. спеціальності 151 «Автоматизація та комп’ютерно-інтегровані технології», освітньо-професійної програми «Комп’ютерно-інтегровані технології та системи навігації і керування» / С. О. Цибульник ; КПІ ім. Ігоря Сікорського. Київ: КПІ ім. Ігоря Сікорського, 2021. 126 с.

Навчальний посібник містить стислі теоретичні відомості, необхідні для виконання конкретних практичних завдань з розробки програмного забезпечення, приклади їх реалізації, завдання для самостійної роботи студентів.

Виконання практичних завдань, пов’язаних з освоєнням принципів об’єктно-орієнтованого програмування (інкапсуляції, спадковості, поліморфізму), а також основних конструкцій мови програмування Java, сприятиме закріпленню, поглибленню та узагальненню теоретичних основ курсу, а також розвитку навичок самостійної творчої роботи студентів у процесі їх навчання, зокрема при виконанні контрольних та інших видів робіт з дисципліни «Технології розробки програмного забезпечення».

ЗМІСТ

ВСТУП……... 4

Комп’ютерний практикум №1. Виведення даних на екран……. 5

Комп’ютерний практикум №2. Робота з класами та методами. Конструктор об’єктів……... 41

Комп’ютерний практикум №3. Введення даних з клавіатури……. 55

Комп’ютерний практикум №4. Використання розгалуження……. 80

Комп’ютерний практикум №5. Робота з циклами…… 98

Комп’ютерний практикум №6. Створення, заповнення, сортування масивів……... 109

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ…… 125

ВСТУП

Навчальний посібник складено відповідно до чинної робочої навчальної програми кредитного модулю навчальної дисципліни «Технології розробки програмного забезпечення - 1» для студентів приладобудівного факультету, які навчаються за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології» за освітньо-професійною програмою «Комп'ютерно-інтегровані технології і системи навігації та керування». Даний навчальний посібник також може використовуватися студентами інших спеціальностей або освітніх програм.

Мета даного навчального видання – допомогти студентам отримати необхідні знання та уміння з об'єктно-орієнтованого програмування, використання архітектурних рішень, патернів та принципів при вирішенні задач розробки програмного забезпечення.

Виконання практичних завдань, пов'язаних з освоєнням принципів об'єктно-орієнтованого програмування (інкапсуляції, спадковості, поліморфізму), а також основних конструкцій мови програмування Java, сприятиме закріпленню, поглибленню та узагальненню теоретичних основ курсу, а також розвитку навичок самостійної творчої роботи студентів у процесі їх навчання, зокрема при виконанні контрольних та інших видів робіт з дисципліни «Технології розробки програмного забезпечення».

Навчальний посібник містить стислі теоретичні відомості, необхідні для виконання конкретних практичних завдань з розробки програмного забезпечення, приклади їх реалізації, завдання для самостійної роботи студентів.

Комп’ютерний практикум №1

Виведення даних на екран

Мета роботи: розглянути основні методи виведення даних у консоль, вивчити особливості об’єднання рядків та виведення відформатованих даних різних типів.

Теоретичні відомості

Ідентифікатори

Ідентифікатори служать для іменування пакетів, класів, інтерфейсів, методів і змінних і інших елементів. Існують спеціальні правила щодо вибору ідентифікаторів, розглянемо їх:

- ідентифікатором може бути будь-яка послідовність маленьких і великих літер, цифр або символів підкреслення ‘_’ і грошової одиниці ‘$’;
- ідентифікатори НЕ повинні починатися з цифри;
- у Java враховується регістр символів;
- допустимі ідентифікатори: MinTemp, sum, x4, $test, my_number;
- неприпустимі ідентифікатори: 3min, min-temp, yes/no;
- не можна використовувати ключові слова (табл. 1.1) в якості ідентифікаторів.

Таблиця 1.1 – ключові слова в Java

abstract	boolean	break	byte	throws
case	catch	char	class	transient
const	continue	default	do	true
double	else	extends	false	try
final	finally	float	for	void
goto	if	implements	import	volatile
instanceof	int	interface	long	while
native	new	null	package	throw
private	protected	public	return	this
short	static	strictfp	super	synchronized
switch				

При іменуванні класів, методів, змінних та інших ідентифікаторів рекомендується дотримуватися наступних правил:

1. Класи та інтерфейси:
   - перша буква в імені повинна бути заголовною;
   - якщо в імені міститься декілька слів, то кожну першу букву в наступних словах також слід робити великою (наприклад, DateTimeFormatter);
   - імена класів слід робити іменниками. Наприклад, Cat, Exam, PrintReader;
   - імена інтерфейсів необхідно давати у формі прикметників: Comparable, Iterable, Navigable.
2. Методи:
   - першу букву потрібно робити рядковою;

- якщо в імені міститься декілька слів, то кожну першу букву в наступних словах варто робити великою;
- імена треба давати у вигляді сполучення дієслів і іменників: getName, doJob, setLastName.

3. Змінні:
   - першу букву потрібно робити рядковою;
   - якщо в імені міститься декілька слів, то кожну першу букву в наступних словах слід робити великою;
   - необхідно привласнювати короткі імена;
   - імена необхідно обирати таким чином, щоб відразу було зрозуміло для чого використовується ця змінна: firstName, buttonHeight.
4. Константи:
   - константи в Java створюються за допомогою зарезервованого слова final;
   - імена констант необхідно задавати тільки заголовними літерами, а слова в імені розділяти знаком підкреслення: MAX_WEIGHT.
5. Пакети:
   - в імені пакета використовуються тільки маленькі букви;
   - для комерційних проектів пакет повинен починатися з ‘com’, потім йде ім'я організації і назва проекту, наприклад, com.example.mypackage.

Коментарі

Коментарі – це блоки коду, які компілятор ігнорує при компіляції. У мові Java вони бувають трьох видів:

- однорядкові коментарі – задаються двома косими рисками ‘//’;
- багаторядкові коментарі – задаються відкриваючою коментар послідовністю символів ‘/*’ та закриваючою багаторядковий коментар послідовністю символів ‘*/’;
- коментарі для документування – задаються відкриваючою коментар послідовністю символів ‘/**’ та закриваючою коментар послідовністю символів ‘*/’.

Коментар для документування – це особливим чином оформлений коментар до об’єкта програми, призначений для використання будь-яким конкретним генератором документації. Від того, який генератор документації застосовується, залежить синтаксис конструкцій, які використовуються в коментарях для документування.

У коментарі для документування може міститися інформація про автора коду, описуватися призначення об'єкта програми, зміст вхідних і вихідних параметрів – для функції/методу/процедури, приклади використання, можливі виняткові ситуації, особливості реалізації.

Роздільники

У мові Java в якості роздільників використовуються наступні символи:

- круглі дужки ‘( )’ – використовуються для передачі списків параметрів у визначеннях і викликах методів, для позначення операції приведення типів і передування операторів у виразах, які використовуються в керуючих операторах;

- фігурні дужки ‘{ }’ – використовуються для вказівки значень масивів, які автоматично ініціалізуються, а також для визначення блоків коду, класів, методів і локальних областей дії;
- квадратні дужки ‘[ ]’ – використовуються для оголошення типів масивів, а також при зверненні до елементів масивів;
- крапка з комою ‘;’ – завершує оператори;
- кома ‘,’ – розділяє послідовний ряд ідентифікаторів в оголошеннях змінних. Застосовується також для створення ланцюгів операторів в операторах циклу;
- крапка ‘.’ – використовується для відділення назв пакетів від підпакетів і класів, а також для відділення змінної або методу від посилальної змінної.

Примітивні типи даних

Типи в Java розподілені на дві категорії (рис. 1.1): примітивні і посилальні (об'єктні). Посилальні типи – це масиви, класи і інтерфейси.

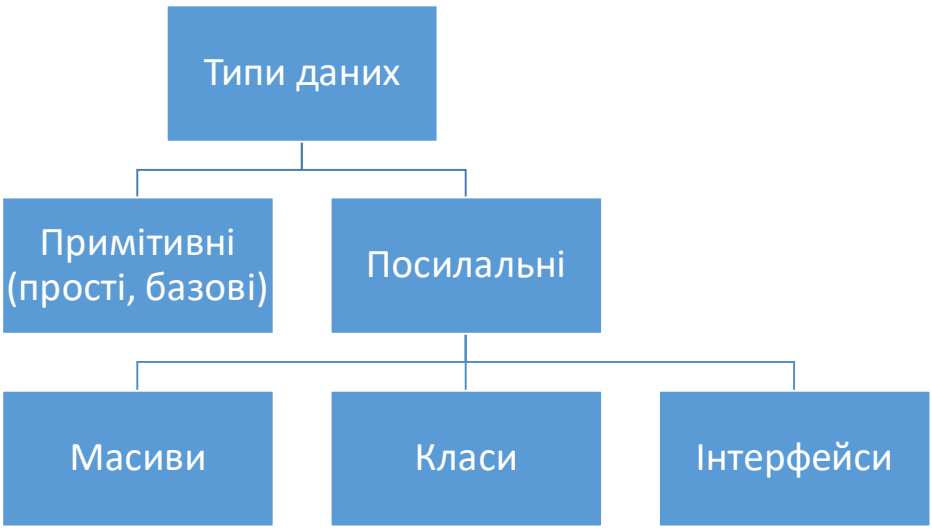

Рисунок 1.1 – Діаграма типів даних в Java

Примітивні типи (рис. 1.2) можна розділити на наступні три групи:

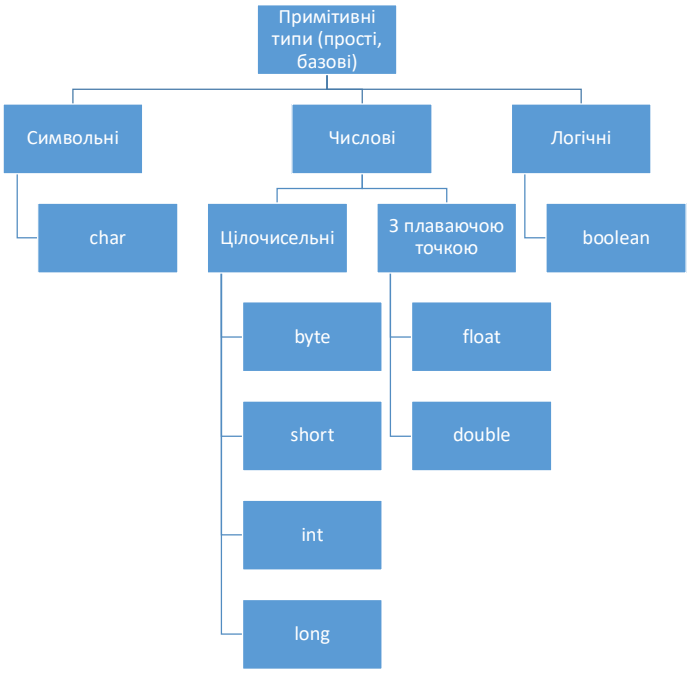

Рисунок 1.2 – Діаграма примітивних типів даних в Java

1. **Цілі числа**. Ця група включає в себе типи даних **byte**, **short**, **int** і **long**, які мають цілі числа зі знаком:

- **int** – основний цілочисельний тип, який використовується в Java за замовчуванням. Будь-яке ціле число буде розглядатися компілятором як число типу **int**;
- **long** – цілочисельний тип, який містить практично нескінченну кількість значень. Використовується у випадках, де числа перевищують 2 мільярди і стандартного **int** вже не вистачає. Використовується в повсякденному житті для створення унікальних значень;
- **byte** – використовується для передачі даних по мережі, запису і читання з файлу. У математичних операціях, як правило, не використовується;

- **short** – самий рідко використовуваний тип в Java, зазвичай використовуватися тільки в цілях економії пам'яті.

2. **Числа з плаваючою крапкою**. Ця група включає в себе типи даних **float** і **double**, що представляють числа з точністю до певного знака після десяткового дробу:

- **double** – це числа з подвійною точністю, максимально наближені до заданих або отриманих в результаті обчислень значень;
- **float** – менш точний тип з плаваючою точкою. Використовується з метою економії пам'яті.

У мові Java є три спеціальні числа плаваючою точкою, які використовуються для позначення переповнення і помилок:

- додатна нескінченність – результат ділення додатного числа на 0. Представлені константами Double.POSITIVE_INFINITY і Float.POSITIVE_INFINITY.
- від'ємна нескінченність – результат ділення від'ємного числа на 0. Представлені константами Double.NEGATIVE_INFINITY і Float.NEGATIVE_INFINITY.
- NaN (англ. not a number, не число) – це, наприклад, обчислення відношення 0/0 або квадратного кореня з від'ємного числа. Представлені константами Double.NaN і Float.NAN.

Числа з плаваючою точкою не можна використовувати в фінансових розрахунках, де помилки округлення недопустимі. Наприклад, у результаті виконання інструкції 'System.out.println (2.0 - 1.1);' буде виведено значення 0.8999999999999999, а не 0.9, як було б логічно припустити. Подібні помилки пов'язані з внутрішнім двійковим поданням чисел. Як в десятковій системі числення не можна точно уявити результат ділення 1/3, так і в двійковій системі неможливо точно уявити результат ділення 1/10. Якщо ж по-

трібно виключити помилки округлення, то слід користуватися класом BigDecimal.

3. **Символи**. Ця група включає в себе тип даних **char**, що представляє символи, наприклад букви і цифри, з певного набору. Символи перетворюються по таблиці кодування UTF-16. Тип **char** є псевдоцілочисельним типом, тому значення цього типу можна, наприклад, задавати у вигляді числа – коду символу з таблиці кодування UTF-16. Кожному символу відповідає певне число з таблиці і Java при знаходженні такого числа в рамках типу **char** виводить його на екран як конкретний символ.

4. **Логічні значення**. Ця група включає в себе тип даних **boolean**, спеціально призначений для представлення логічних значень. Логічні змінні цього типу можуть приймати тільки два значення: true – істина і false – брехня.

Таблиця 1.2 містить інформацію про довжину, діапазон допустимих значень і значення за замовчуванням примітивних типів.

Таблиця 1.2 – Примітивні типи даних

Тип	Розмір в байтах	Значення від..до	Значення за замовчуванням
boolean	—	true або false	false
byte	1	-128..127	0
short	2	-32,768..32,767	0
int	4	-2,147,483,648..2,147,483,647	0
long	8	-9,223,372,036,854,775,808.. 9,223,372,036,854,775,807	0

Продовження таблиці 1.2

char	2	0..65,535	'\u0000'
float	4	-3.4E+38..3.4E+38 (стандарт IEEE754)	0.0
double	8	-1.7E+308..1.7E+308 (стандарт IEEE754)	0.0

Літерали

Літерал – це синтаксичний елемент, який представляє значення. Тобто літерали – це явно задані значення в коді програми, наприклад, 1, 7.666f, false, 'R', "Hello world\n", тощо.

Цілочисельні літерали. Типи цілочисельних літералів в Java:

• десяткові;

• вісімкові;

• шістнадцяткові;

• двійкові (починаючи з Java 7).

Усі цілочисельні літерали представляють значення **int**. Літерал типу **int** може присвоюватися змінним типів **byte** та **short** без приведення типів, якщо значення літерала знаходиться в межах допустимих значень даних типів. Для створення літерала типу **long**, необхідно явно вказати компілятору тип, доповнивши літерал буквою 'l' або 'L' (240437L, 0x4FFl, тощо).

Десяткові літерали – 88, +88, -88.

Вісімкові літерали починаються з 0 і використовують числа від 0 до 7, наприклад:

06 // Відповідає десятковому числу 6

07 // Відповідає десятковому числу 7

010 // Відповідає десятковому числу 8

011 // Відповідає десятковому числу 9

Шістнадцяткові літерали утворюються за наступними правилами:

- складаються з символів 0-9, a-f, A-F;
- повинні починатися з 0x або 0X;
- мають обмеження по довжині до 16 символів, не враховуючи префікс 0x і необов'язковий суфікс L.

Приклади шістнадцяткових літералів: 0X0105, 0x7ooffoff, 0xccccccccc, тощо.

Двійкові літерали складаються з комбінації цифр 0 та 1. Також для визначення двійкового літерала необхідно додати префікс 0b або 0B до числа.

Приклади двійкових літералів: 0b101, 0B101, тощо.

Літерали з плаваючою точкою. Усім літералам з плаваючою точкою за замовчуванням присвоюється тип **double**. Щоб створити літерал типу **float**, потрібно після літерала вказати букву 'f' або 'F'.

Приклади літералів з плаваючою точкою: 4345.545463F, 178.15f, 1454.6767, .4 (число 0.4), тощо.

До літералів типу **double** можна додавати символи 'D' або 'd', але це не обов'язково (15.454545D, 151.12d, тощо).

Цілочисельні літерали та літерали з плаваючою точкою можна представити у експоненційній формі. Для цього до стандартної форми літерала додаються символи 'e' або 'E', далі записується число, яке є ступенем числа 10. Також ступінь може містити знаки '+' або '-'.

Приклади експоненціальних літералів: 1E+10, 1.9e-12, 1E14, тощо.

Підкреслення в числових літералах. У Java існує можливість використовувати будь-яку кількість символів підкреслення для поділу груп цифр, що покращує читабельність числа в цілому. Підкреслення може роз-

діляти тільки цифри. Не можна використовувати підкреслення в наступних місцях:

- на початку або вкінці числа;
- поруч з десятковою крапкою в числі з плаваючою точкою;
- перед суфіксами 'F', 'f', 'l' або 'L';
- для поділу символів префіксів.

Приклади правильного використання підкреслення в літералах: 12_567, 56.356_234, 0b000____0100, тощо.

Приклади неправильного використання підкреслення в літералах: 220_, 0_x2F, 0x_3E, 6345.56_f, 3423_.87f, тощо.

Літерали типу boolean. Значення літерала типу **boolean** може бути визначено лише як true або false.

Символьні літерали. Символьні літерали – це символи, які підтримують набір символів Unicode. Для представлення символьних літералів в Java використовується тип даних **char**.

Символьні літерали мають бути представлені у вигляді символу розміщеного в одинарних лапках, наприклад, 'n', '#'.

Є можливість присвоєння числового літерала символьному типу. Числовий літерал повинен знаходитися в діапазоні від 0 до 65535. Наприклад:

char a1 = 0x615; // шістнадцятковий цілочисельний літерал

char a2 = 841; // десятковий цілочисельний літерал

char a3 = (char) 70000; // Робиться приведення типів, тому що 80000 перевищує діапазон від 0 до 65535

Крапка з комою в кінці кожного рядка є необхідною. Нею закінчується кожна інструкція на мові Java.

Рядкові літерали. Рядкові літерали належать об'єктам типу **String**, який являється одним з *класів-оболонок*. Більш детально про класи-оболонки буде пояснено у наступних роботах.

Є декілька основних правил використання рядкових літералів, а саме:

- вони розташовуються між двома подвійними апострофами;
- керуючі символи, а також вісімкова та шістнадцяткова форми запису, що використовується в символьних літералах, діють точно так же і в рядкових літералах;
- рядкові літерали можуть розташовуватися тільки на одному рядку вихідного коду.

Приклади рядкових літералів:

String str1 = "MyProgram";

String str2 = "first line \n second line";

String str4 = "\u004F letter"; // O letter

Методи класу рядків наведено в табл. 1.3.

Таблиця 1.3 – Методи класу String

№	Метод (аргументи)	Величина, яку повертає метод	Опис методу
1	2	3	4
1	charAt (int index)	char	Повертає символ за вказаним індексом
2	compareTo (Object o)	int	Порівнює даний рядок з іншим об'єктом

Продовження таблиці 1.3

1	2	3	4
3	compareTo (String anotherString)	int	Порівнює два рядки лексично
4	compareToIgnoreCase (String str)	int	Порівнює два рядки лексично, ігноруючи регістр букв
5	concat (String str)	String	Об'єднує початковий рядок з рядком-аргументом, шляхом додавання останнього в кінці
6	contentEquals (StringBuffer sb)	boolean	Повертає значення true тільки в тому випадку, якщо початковий рядок являє собою ту ж послідовність символів, які вказано в аргументі
7	static copyValueOf (char[] data)	String	Повертає рядок, який представляє собою послідовність символів з зазначеного масиву
8	static copyValueOf (char[] data, int offset, int count)	String	Повертає рядок, який представляє собою послідовність символів з зазначеного масиву
9	endsWith (String suffix)	boolean	Перевіряє чи закінчується початковий рядок зазначеним закінченням
10	equals (Object anObject)	boolean	Порівнює даний рядок із зазначеним об'єктом

Продовження таблиці 1.3

1	2	3	4
11	equalsIgnoreCase (String anotherString)	boolean	Порівнює даний рядок з іншого рядком, ігноруючи регістр букв
12	getBytes ()	byte	Кодує рядок в послідовність байтів за допомогою платформи charset
13	getBytes (String charsetName	byte[]	Кодує рядок в послідовність байтів за допомогою платформи charset, зберігаючи результат у новий масив байтів
14	getChars (int srcBegin, int srcEnd, char[] dst, int dstBegin)	void	Копіює символи з поточного рядка в масив символів
15	hashCode ()	int	Повертає хеш-код поточного рядка.
16	indexOf (int ch)	int	Повертає індекс першого входження зазначеного символу в поточному рядку
17	indexOf (int ch, int fromIndex)	int	Повертає індекс першого входження зазначеного символу в поточному рядку, починаючи пошук з зазначеного індексу
18	indexOf (String str)	int	Повертає індекс першого входження зазначеного підрядка в поточному рядку

Продовження таблиці 1.3

1	2	3	4
19	indexOf (String str, int fromIndex)	int	Повертає індекс першого входження зазначеного підрядка в поточному рядку, починаючи з зазначеного індексу
20	intern ()	String	Повертає канонічне уявлення для рядкового об'єкта
21	lastIndexOf (int ch)	int	Повертає індекс останнього входження зазначеного символу в поточному рядку
22	lastIndexOf (int ch, int fromIndex)	int	Повертає індекс останнього входження зазначеного символу в поточному рядку, починаючи зворотний пошук з зазначеного індексу
23	lastIndexOf (String str)	int	Повертає індекс останнього входження зазначеного підрядка в поточному рядку
24	lastIndexOf (String str, int fromIndex)	int	Повертає індекс останнього входження зазначеного підрядка в поточному рядку, починаючи зворотний пошук з зазначеного індексу
25	length ()	int	Повертає довжину рядка

Продовження таблиці 1.3

1	2	3	4
26	matches (String regex)	boolean	Повідомляє, чи відповідає поточний рядок заданому регулярному виразу
27	regionMatches (boolean ignoreCase, int toffset, String other, int ooffset, int len)	boolean	Перевіряє чи рівні дві області рядка
28	regionMatches (int toffset, String other, int ooffset, int len)	boolean	Перевіряє чи рівні дві області рядка
29	replace (char oldChar, char newChar)	String	Повертає новий рядок, замінивши у результаті всі входження oldChar у поточному рядку на newChar
30	replaceAll (String regex, String replacement)	String	Замінює у поточному рядку кожен підрядок, який відповідає regex, на вираз replacement
31	replaceFirst (String regex, String replacement)	String	Замінює у поточному рядку перший підрядок, який відповідає regex, на вираз replacement
32	split (String regex)	String[]	Розділяє поточний рядок
33	split (String regex, int limit)	String[]	Розділяє поточний рядок

Продовження таблиці 1.3

1	2	3	4
34	startsWith (String prefix)	boolean	Перевіряє, чи починається поточний рядок з заданого префікса
35	startsWith (String prefix, int toffset)	boolean	Перевіряє, чи починається поточний рядок з зазначеного префікса, починаючи з зазначеного індексу
36	subSequence (int beginIndex, int endIndex)	CharSequence	Повертає нову послідовність символів, яка є підпослідовністю поточної послідовності
37	substring (int beginIndex)	String	Повертає новий рядок, який є підрядком поточного рядка
38	substring (int beginIndex, int endIndex)	String	Повертає новий рядок, який є підрядком поточного рядка
39	toCharArray ()	char[]	Перетворює поточний рядок в новий масив символів
40	toLowerCase ()	String	Перетворює всі символи в поточному рядку в нижній регістр
41	toLowerCase (Locale locale)	String	Перетворює всі знаки в поточному рядку в нижній регістр, використовуючи правила мовного стандарту
42	toString ()	String	Цей об'єкт (який вже є рядком!) повертає себе

Продовження таблиці 1.3

1	2	3	4

43	toUpperCase ()	String	Перетворює всі символи в рядку у верхній регістр
44	toUpperCase (Locale locale)	String	Перетворює всі символи в рядку у верхній регістр, використовуючи правила мовного стандарту
45	trim ()	String	Повертає копію рядка з пропущеними початковими і кінцевими пробілами
46	static valueOf (primitive data type x)	String	Повертає рядкове представлення переданого в аргументі примітивного типу даних

Керуючі символи та послідовності

Є безліч символів, які з різних причин не можна представити безпосередньо. У цьому випадку необхідно використовувати керуючі послідовності – спеціальні символьні комбінації, які використовуються у функціях введення і виведення інформації. Символи, з яких складаються керуючі послідовності, називаються керуючими – це символи, які не мають графічного відображення, але використовуються для керування пристроями, організації даних або інших цілей.

Існує декілька керуючих символів, які вказують компілятору про початок керуючої послідовності. Найчастіше керуюча послідовність починається з використання зворотної дробової риски '\', за якою розміщується комбінація латинських букв і цифр. Перелік найбільш поширених керуючих послідовностей, які використовуються в символьних та рядкових літералах, представлено в табл. 1.4.

Таблиця 1.4 – основні керуючі послідовності

Керуюча послідовність	Опис
\a	Звуковий сигнал
\b	Повернення на один символ назад. Дозволяє видалити останній символ в рядку виводу, подібно клавіші backspace
\t	Символ табуляції є аналогом чотирьох пробілів. Символ табуляції часто використовується для побудови таблиць або псевдографічних елементів інтерфейсу, тому що це зручніше, ніж запис пробілів
\n	Перехід на новий рядок
\v	Вертикальна табуляція
\r	Повернення курсору на початок поточного рядка і відображення нової інформації так, ніби раніше в цьому рядку нічого не було
\"	Подвійний апостроф
\'	Апостроф
\0	Нуль-символ
\\	Обернена коса риска
\OOO	Вісімковий код ASCII або ANSI символу
\xHHHH	Шістнадцятковий код ASCII або ANSI символу

Керуючі послідовності \OOO і \xHHHH (О позначає вісімкову цифру; Н позначає шістнадцяткову цифру) дозволяють представити символ з таб-

лиці ASCII або ANSI як послідовність вісімкових або шістнадцяткових цифр відповідно, наприклад:

'\u004F' // Буква 'O' в шістнадцятковій формі;

'\141' // Буква 'a' в вісімковій формі.

Необхідно відзначити той факт, що, якщо зворотна дробова риска передує символу, який не є керуючим або цифрою, то це призведе до помилки компіляції.

Екранування символів. Як показано вище, на мові Java текст можна представити у вигляді літералів типу **String**, для позначення даних якого використовуються керуючі символи – парні подвійні апострофи (подвійні верхні лапки), наприклад, "Hello World!".

З даним текстом ніяких проблем не виникає, але якщо цей же текст необхідно виділити прямою мовою, то, скориставшись правилами грамматики, стає ясно, що текст "Hello World!", крім керуючих символів від типу **String**, потрібно помістити в лапки прямої мови, наприклад, "Прокидаючись вранці, скажіть "Hello World!" ". Такий варіант викличе помилку компіляції, оскільки компілятор не зможе зрозуміти, коли насправді закінчується приведений рядковий літерал.

Для вирішення цієї та подібних їй проблем було придумано екранувати символи, тобто міняти певні символи на керуючі послідовності, відомі також, як *escape-послідовності*. При використанні керуючих послідовностей з табл. 1.4 отримаємо рядковий літерал, який не викличе проблем під час компіляції: "Прокидаючись вранці, скажіть \"Hello World!\" ".

Змінні: оголошення та ініціалізація

Змінна – це іменована область пам'яті для зберігання даних, які можуть змінюватися в процесі виконання програми.

Кожна змінна характеризується:

- ідентифікатором (‘позначенням комірки пам'яті’);
- значенням (даними, що містяться в змінній в конкретний момент часу);
- Java – строго типізована мова. Кожна змінна в Java має конкретний тип, який визначає розмір області пам’яті для її розміщення; діапазон значень, які можуть зберігатися в пам'яті; набір операцій, які можуть бути застосовані до змінної;
- перед використанням змінну необхідно оголосити;
- якщо потрібно оголосити декілька змінних заданого типу, це можна зробити однією інструкцією у вигляді розділеного комами списку ідентифікаторів змінних;
- змінна може бути ініціалізована при оголошенні або пізніше;
- під час виконання програми значення змінної може змінюватися;
- вираз ініціалізації має повертати значення того ж самого (або сумісного) типу, що й у змінної;
- ідентифікатор та тип даних змінної не можуть змінюватися в процесі виконання програми.

Отже, перед використанням, кожна змінна повинна бути оголошена. Оголошення змінної – це вибір для неї унікального ідентифікатора та присвоювання певного типу даних. Загальна форма оголошення змінної має вигляд:

тип_даних ідентифікатор;

де *тип_даних* позначає один з примітивних типів даних в Java, ім'я класу або інтерфейсу. А *ідентифікатор* – це ім'я змінної.

Приклади оголошення змінних:

int a;

byte t;

double pi;

Ініціалізація змінної – це присвоювання змінній певного значення, яке відповідає її типу даних. Це означає, що в загальному випадку ініціалізація має відбуватися після оголошення змінної. Ініціалізація оголошеної змінної відбувається наступним чином:

ідентифікатор = значення;

Приклади ініціалізації змінних:

x = 0D;

b = true;

c = 'A';

При *динамічній ініціалізації* змінній присвоюється значення у вигляді деякого виразу, що є дійсним у даний момент, наприклад, p = (a+b+c)/2.

У мові Java існує три типи змінних:

- локальні змінні:
    - оголошуються в методах, конструкторах або блоках;
    - створюються, коли метод, конструктор або блок запускається і знищуються після того, як завершиться метод, конструктор або блок;
    - можна використовувати модифікатори доступу ;
    - є видимими тільки в межах оголошеного методу, конструктора або блоку (локальна область видимості);

- реалізуються в середині стеку.

- змінні екземпляра:
  - оголошуються в класі, але за межами будь-якого методу, конструктора або блоку.
  - створюються тоді, коли за допомогою ключового слова *new* створюється об’єкт і видаляються, коли відповідний об’єкт знищується;
  - містять значення, які можуть використовуватися більш ніж одним методом, конструктором або блоком коду;
  - Змінні примірника можуть бути оголошений на рівні класу, до або після використання.
  - описують стан об’єкта, його властивості або характеристики;
  - можуть мати модифікатори доступу;
  - є видимими для всіх методів, конструкторів і блоків в класі. Рекомендується застосовувати їх з модифікатором доступу *private*. Однак також можна зробити їх видимими для підкласів;
  - мають значення за замовчуванням. Для чисел за замовчуванням дорівнює 0, для логічних величин – false, для посилань на об’єкт – null. Нові значення можуть бути присвоєні, наприклад, при оголошенні або в конструкторі;
  - можуть бути доступні безпосередньо шляхом виклику імені змінної всередині класу. Однак в статичних методах та інших класах (коли до змінних екземпляра є доступ завдяки відповідним модифікаторам) повинні бути

викликані використовуючи повну інструкцію виклику – **Ім’я_об’єкта.Ім’я_змінної_екземпляра**.

- статичні змінні або змінні класу:
    - також відомі в Java як статичні змінні. Вони оголошуються з ключовим словом *static*. Оголошення проходить в межах класу, але за межами методу, конструктора або блоку коду;
    - незалежно від того, скільки об’єктів даного класу створено вони усі будуть використовувати одну спільну статичну змінну для опису стану, властивості або характеристики;
    - майже не використовуються, часто оголошуються як константи – змінні, які оголошені з ключовим словом *final*. Константи ніколи не змінюють свого початкового значення;
    - створюються при запуску програми і знищуються, коли виконання програми завершується;
    - більшість статичних змінних мають модифікатор доступу public, оскільки вони повинні бути доступні для користувачів класу;
    - значення за замовчуванням таке ж, як і у змінних екземпляра. Значення можуть бути присвоєні, наприклад, при оголошенні або в конструкторі. Крім того, значення можуть бути присвоєні в статичних методах або блоках коду;
    - можуть бути викликані за допомогою інструкції – **Ім’я_класу.Ім’я_змінної**;

- при оголошенні з набором модифікаторів (одночасно) *public*, *static* та *final* імена змінних класу необхідно задавати в верхньому регістрі.

Виведення даних

Оскільки Java є об'єктно-орієнтованою мовою програмування, то всі файли являють собою класи. Для звернення до класів необхідно створювати об'єкти і через об'єкти можна отримувати дані з класу. Для виведення даних на екран спочатку необхідно звернутися до класу System, після чого взяти його об'єкти і методи. Оскільки в System є створений об'єкт out, а у нього є методи print() і println(), які призначені для виведення даних в консоль, можна звернутися до них, скориставшись наступним виразом:
System.out.print ("Hello World!"); або *System.out.println ("Hello World!");*

Подібна структура присутня в усій мові Java.

Різниця між використаними методами полягає у тому, що метод print() виводить текст без пропуску рядка, а метод println() виводить текст і після цього переводить курсор на наступний рядок. За необхідності можна самостійно вказати пропуск рядка при виведенні тексту, скориставшись керуючими послідовностями:

System.out.print ("Hello \ n \ n \ nWorld!");

Конкатенація рядків

Конкатенація рядків (зчеплення) – це операція об'єднання рядків. Для цієї операції в Java використовується знак +. До рядку можна приєд-

нувати не тільки інший рядок, а й значення будь-якого іншого типу, яке буде перетворено до рядка автоматично.

Приклад об'єднання двох рядків:

```
String str = "world!";
System.out.println("Hello " + str);
System.out.println("Перший рядок\n" + "Другий рядок");
```

Другий приклад об’єднання рядків:

```
char x, y;
x = 88;  // Код символа 'X'
y = 'Y';
System.out.println(x + " " + y); //Виведеться X Y
System.out.println(x + y); //Виведеться 177
```

У першому System.out.println() на консоль виведеться рядок "X Y". Відбувається об'єднання char зі String – це конкатенація. У другому ж System.out.println() на консоль виведуться не рядки, а число 177. Тип **char** є псевдоцілочисельним типом, і може брати участь в арифметичних операціях. У даному випадку складаються коди символів X та Y.

Виведення відформатованих даних

У ряді випадків розглянутий вище метод виведення даних викликає ускладнення. Так, якщо вивести на екран велике число, то велика кількість цифр ускладнить його сприйняття. Тому у Java був реалізований метод printf(), який дозволяє форматувати рядки і має спрощену форму запису. Нижче наведено приклад виклику цього методу з декількома параметрами:

```
System.out.printf ( "Hello,%s. Next year, you'll be %d", name, age);
```

Як видно, наведений рядок формату містить простий текст і два правила форматування. Перше правило використовується для форматування рядкового аргументу, а друге для форматування цілочисельного типу. Загальний вигляд конструкції специфікатора формату наступний:

%[флаги][ширина][.точність]символ_перетворення

де:

- % – керуючий символ, що позначає початок інструкцій форматування;
- [флаги] – спеціальні символи для форматування. Не є обов'язковою частиною інструкції. Усі флаги приведені в табл. 1.5;
- [ширина] – позитивне ціле десяткове число, яке визначає мінімальну кількість символів, які будуть виведені. Не є обов'язковою частиною інструкції.
- [.точність] – не від’ємне ціле десяткове число з крапкою перед ним. Зазвичай використовується для обмеження кількості символів, які будуть виводитися на екран. Специфіка поведінки залежить від виду перетворення. Не є обов'язковою частиною інструкції.
- символ_перетворення – це символ, який вказує, як повинен бути відформатований аргумент. Усі символи перетворення приведені в табл. 1.6.

Розглянемо приклади використання специфікаторів формату.

Щоб розбити рядок на окремі рядки, існує специфікатор %n:

System.out.printf ("перший %n другий %n третій");

Наведений вище фрагмент коду дасть наступний результат:

перший

другий

третій

Таблиця 1.5 – Флаги специфікатора формату

Флаг	Призначення
+	Виводить знак для додатних та від'ємних чисел
Пробіл	Додає пробіл перед додатними числами
0	Виводить початкові нулі
-	Вирівнює по лівому краю
(	Розміщує від'ємні числа у круглих дужках
,	Задає використання роздільника груп, наприклад, –3,765.29
# (для формату f)	Завжди відображає десяткову точку
# (для формату x)	Додає префікс 0x, наприклад, 0x1234ffff
$	Указує позицію аргументу у списку аргументів. Посилання на перший аргумент – 1$, посилання на другий аргумент – 2$ і так далі. Якщо позиція явно не задана, то аргументи повинні знаходитися в тому ж порядку, що і посилання на них в рядку форматування
<	Задає форматування того ж самого значення, яке відформатовано попереднім специфікатором формату. Наприклад, вираз %f %<x указує на те, що одне і теж саме число має бути представлене як у вигляді з плаваючою точкою, так і у шістнадцятковій формі

Таблиця 1.6 – Символи перетворення для методу printf ()

Символ перетворення	Тип
d	Десяткове ціле число
x	Шістнадцяткове ціле число
f	Число з фіксованою або плаваючою точкою
e	Число з плаваючою точкою в експоненціальній формі
g	Число з плаваючою точкою в загальному форматі. Майже не використовується
a	Шістнадцяткове число з плаваючою точкою
s	Символьний рядок
c	Символ
b	Логічне значення
h	Хеш-код
Tx або tx	Дата та час
%	Знак проценту
n	Роздільник рядків, який залежить від платформи

Для форматування логічних значень використовується специфікатор %b:

System.out.printf ("%b%n", null);

System.out.printf ("%B%n", false);

System.out.printf ("%B%n", 5.3);

System.out.printf ("%b%n", "random text");

У результаті буде отримано:

false

FALSE

TRUE

true

Це працює наступним чином: якщо вхідне значення true, то вихідне значення буде також true. В іншому випадку на виході буде false. Варто зауважити, що можна використовувати, наприклад, %B для форматування в верхньому регістрі.

Для форматування рядків необхідно використовувати комбінацію %s:

printf ("'% s'% n", "рядок");

printf ("'% S'% n", "рядок");

У результаті буде отримано:

рядок

РЯДОК

Більш того, можна обмежити кількість символів у виведенні, вказавши точність:

System.out.printf ("%.4s", "Олександр!");

У результаті буде отримано:

Олек

Результатом використання специфікатора %c є символ Unicode:

System.out.printf ("%c%n", 's');

System.out.printf ("%C%n", 's');

s

S

Форматування дати і часу. Для форматування дати і часу рядок перетворення являє собою послідовність з двох символів: символу 't' або 'T' і суфікса перетворення.

Розглянемо найбільш поширені символи суфіксів форматування дати і часу на прикладах. Слід зауважити, що для більш складного форматування можна використовувати клас **DateTimeFormatter**.

Форматування часу. Список корисних символів суфікса для форматування часу:

- H – друкує годину дня для від 00 до 23;
- I – друкує годину дня для від 01 до 12;
- k – друкує годину дня для без ведучого 0 у діапазоні 0-23;
- l – друкує годину дня без ведучого 0 у діапазоні 1-12;
- M – друкує хвилини у діапазоні від 00 до 59.
- N – друкує наносекунди, відформатовані з 9 цифрами і початковими 0 у діапазоні від 000000000 до 999999999;
- Q – друкує мілісекунди з 1 січня 1970 р 00:00:00 UTC;
- R – друкує час відформатований як години:хвилини у діапазоні до 23 годин;
- r – друкує час відформатований як %tI:%tM:%tS %tp;
- S – друкує секунди протягом хвилини, відформатовані за 2 цифрами від 00 до 60. 60 потрібно для підтримки високосних секунд;
- p – друкує специфічний для локалі маркер ‘am’ або ‘pm’;
- s – друкує секунди з 1 січня 1970 р 00:00:00 UTC;
- T – друкує час відформатований як %tI:%tM:%tS;
- Z – друкує Скорочення часового поясу, наприклад ‘UTC’, ‘PST’, тощо;
- z – друкує зміщення часового поясу від GMT, наприклад, -0800.

Припустимо, що треба роздрукувати часову частину Date:

Date date = new Date ();

System.out.printf ("%tT%n", date);

Наведений вище код разом з комбінацією %tT дає наступний результат:

17:22:45

У разі, якщо потрібно більш детальне форматування, можна викликати різні часові сегменти:

System.out.printf ("hours%tH: minutes%tM: seconds%tS%n", date, date, date);

Скориставшись 'H', 'M' і 'S', буде отримано:

hours 13: minutes 51: seconds 15

Багаторазове використання змінної date не є раціональним. В якості альтернативи, щоб позбутися від декількох однакових аргументів, можна використовувати індексне посилання на вхідний параметр. Для розглянутого випадку – 1$, що перетворить інструкцію в коректну форму:

System.out.printf ("%1$tH:%1$tM:%1$tS%1$tp%1$tL%1$tN%1$tz%n", date);

У даному прикладі буде виведено поточний час, am/pm, час у мілісекундах, наносекундах і зміщення часового поясу:

13:51:15 pm 022 047 000 000 +0400

Форматування дати. Як і для форматування часу, є спеціальні символи для форматування дати:

- A – друкує повну назву дня тижня;
- a – друкує скорочену назву дня тижня;
- B – друкує повну назву місяця;
- b – друкує скорочену назву місяця;
- C – друкує частину року, яка відповідає за століття, та відформатовує двома цифрами від 00 до 99;
- c – друкує дату і час в форматі ‘%ta % tb% td% tT% tZ% tY’;

- D – друкує дату в форматі ‘%tm/%td/%ty’;
- d – друкує день місяця в форматі двох цифр від 01 до 31;
- e – друкує день місяця, відформатований без першого 0 від 1 до 31;
- F – друкує дату в форматі ISO 8601 з ‘%tY-%tm-%td’;
- h – друкує те ж, що і %tb.
- j – друкує день року, відформатований з початковими 0 від 001 до 366;
- m – друкує місяць відформатований з початковим 0 від 01 до 12.
- Y – друкує рік відформатований 4 цифрами від 0000 до 9999;
- y – друкує рік відформатований двома цифрами від 00 до 99.

Отже, якщо необхідно показати день тижня, а потім місяць:

System.out.printf ("%1$tA,%1$tB%1$tY%n", date);

У результаті буде отримано:

Thursday, November 2018

Щоб усі результати були представлені в числовому форматі, можна замінити літери ‘A’, ‘B’, ‘Y’ на ‘d’, ‘m’, ‘y’:

System.out.printf ("%1$td.%1$tm.%1$ty%n", date);

Що призведе до результату:

15.10.20

Порядок виконання роботи.

Основні завдання, які необхідно вирішити: ознайомитися з програмним забезпеченням IntelliJ IDEA, навчитися оголошувати та ініціалізувати змінні різних типів та виводити дані у консоль.

Розв'язання: Для рішення перелічених завдань необхідно скористатися програмним пакетом IntelliJ IDEA, який має модульну структуру, тобто складається з декількох частин, кожна з яких має власне функціональне призначення. Основним модулем, роботу з яким буде розглянуто у циклі комп'ютерних практикумів є проект Java. Даний модуль дозволяє проводити розробку програмного забезпечення, використовуючи Java Core.

Скориставшись IntelliJ IDEA, необхідно навчитися оголошувати та ініціалізувати змінні різних типів, а також виводити на екран інформацію, яка міститься у зазначених змінних, за допомогою методів print(), println() та printf(). Необхідно порівняти функціональні можливості виведення даних з використанням конкатенації рядків та відформатованого виведення. Увесь необхідний теоретичний матеріал розглянуто вище.

Звіт з практикуму має містити: стислі теоретичні відомості, умову завдання, код програми з необхідним описом, результати виконання, висновки по роботі.

Завдання для виконання.

Частина 1. За допомогою керуючих послідовностей (наприклад, **\n, \t** або **%10**) та використовуючи методи **print()**, **println()** та **printf()** вивести на екран таблицю, яка складається мінімум з 5 колонок та 8 рядків, а також містить 6 різних типів даних. Наповнення таблиці (орієнтовний перелік тем наведено в табл. 1.7) має бути унікальним для кожного студента.

Приклад:

Ім'я	Прізвище	Студент	Коефіцієнт	Бал
Петров	Олег	true	1,10	77,27*1,10=85(B)
Сидоров	Василь	true	1,17	64,10*1,17=75(C)
Іванов	Олександр	false	1,0	30*1,00=30(F)
Абрамов	Ігор	true	1,17	81,20*1,17=95(A)
Петров	Олександр	false	1,0	30*1,00=30(F)
Іванов	Василь	false	1,0	30*1,00=30(F)
Сидоров	Ігор	true	1,17	64,10*1,17=75(C)
Абрамов	Олег	true	1,10	77,27*1,10=85(B)

Таблиця 1.7 – Теми для завдань

Тварини	Книги	Космос	Одяг	Іграшки
Магазин	Освіта	Друзі	Продукти	Дерева
Кіно	Фільми	Телефони	Автомобілі	Відпочинок
Квіти	Косметика	Ігри	Меблі	Подарунки
Мотоцикли	Час	Медицина	Серіали	Програмування
Техніка	Хобі	Спорт	Промисловість	Ґаджети

Частина 2. Створіть декілька змінних, які містять будь-які текстові рядки, наприклад, "*I like Java !!!*", "*Я вивчаю Java!*", тощо. Для даних рядків виконати:

а) Роздрукувати останній символ рядка. Використати метод **String.charAt()**.

б) Перевірити, чи закінчується рядок підрядком "*!!!*". Використати метод **String.endsWith()**.

в) Перевірити, чи починається рядок підрядком "*I like*". Використати метод **String.startsWith()**.

г) Перевірити, чи містить рядок підрядок "*Java*". Використати метод **String.contains()**.

д) Замінити всі символи "*a*" на "*o*".

е) Вирізати рядок "*Java*" за допомогою методу **String.substring()** таким чином, щоб на екран вивелася фраза без нього.

Ускладнене завдання (за вибором). Є масив даних розміром **ExH** (не менше **8x5**). У циклі необхідно якісно та адекватно сформувати таблицю за допомогою виведення відформатованих даних (метод **printf()**) таким чином, щоб довжина кожної колонки дорівнювала довжині найбільшого за довжиною елементу масиву **+6** символів (по **+3** з кожного боку). Розділювальними знаками обрати для колонок – "|", рядків – "———". Розділюва-

льні знаки для рядків можна використовувати тільки для зовнішніх контурів таблиці та її шапки. Для формування відступів у комірках таблиці використати керуючу послідовність у вигляді специфікатора формату, наприклад, **%15s**, де % – керуючий символ; 15 – кількість символів.

Контрольні запитання.

1. Що таке змінна?
2. Які є типи даних?
3. Чим відрізняється символьний тип даних від рядкового?
4. Чим відрізняється літерал від типу даних?
5. Які є методи виведення даних у консоль?
6. Чим відрізняється оголошення змінної від її ініціалізації? Чи можна виконати оголошення та ініціалізацію змінної однією інструкцією?
7. Які є основні специфікатори при виведенні відформатованих даних?
8. Що таке керуюча послідовність та керуючий символ?

Комп’ютерний практикум №2

Робота з класами та методами. Конструктор об’єктів

Мета роботи: вивчити основи об’єктно-орієнтованого програмування, вивчити та навчитися використовувати основні принципи об’єктно-орієнтованого програмування, навчитися створювати об’єкти та керувати їхньою поведінкою.

Теоретичні відомості

Класи

Об’єктно-орієнтоване програмування – це підхід до розробки додатків за допомогою класів і об'єктів. Він дозволяє писати значно менше коду і при цьому реалізовувати більше можливостей, ніж при функціональному програмуванні.

Клас в Java – це шаблон для створення об’єкта. При використанні класів у програмах виділяються дві складові: оголошення класу як окремого типу даних, а також реалізація об’єкта даного класу.

Клас визначається за допомогою ключового слова class та :

```
class Student {
    // Тіло класу
}
```

Об’єкти дуже часто називають екземплярами відповідних класів, а будь-яку програму на Java можна уявити як набір взаємодіючих між собою об’єктів. При цьому клас визначає поведінку об’єктів (за допомогою мето-

дів) і стан, характеристики або властивості об’єктів (за допомогою змінних екземпляра).

Клас може мати будь-яку кількість змінних та методів.

Методи

Методи в Java – це закінчена послідовність дій (інструкцій), спрямованих на вирішення окремого завдання. Методи, по своїй суті, – це функції (або процедури чи підпрограми більш ранніх мов, які не є об’єктно-орієнтовними), які є членами класів і характеризують поведінку об’єктів.

Загальна форма визначення методу виглядає наступним чином:

```
[модифікатори] тип_даних_для_повернення ім’я_методу(аргументи) {
        // Тіло методу
    }
```

Модифікатори та аргументи не є обов’язковими елементами оголошення методів.

Методи визначаються завжди всередині класів:

```
public class Main {
    public static void test() {
        // Тіло методу
    }
}
```

де test() – це метод, який визначається в класі **Main**; public – модифікатор доступу (табл. 2.1); static – модифікатор, який означає що метод статичний, тобто він належить до класу **Main**, а не конкретному екземпляру класу **Main**; void означає, що цей метод не повертає значення.

Методи можуть повертати значення в Java. Тип даних значення, яке буде повернене після завершення роботи методу, повинен бути визначений при оголошенні методу. Термін "повернути значення" означає, що після завершення методу буде збережено посилання на певний локальний вираз, який можна буде зберегти та використати у іншій частині програми, незважаючи на обмеження, що накладаються на локальні вирази (знищення після виконання локального блоку коду).

Таблиця 2.1 – Модифікатори доступу

Модифікатори доступу	Тіло класу	Пакет, який містить клас	Клас-спадкоємець (підклас)	Інші частини програми, крім названих (наприклад, інші пакети)
public	+	+	+	+
private	+	+	+	-
default (автоматично використовується, коли немає жодного з інших модифікаторів)	+	+	-	-
protected	+	-	-	-

Якщо метод має повертати значення, то його тип указується при оголошенні методу, а в тілі методу повинен бути хоча б один оператор

return вираз_який_буде_повернено;

де тип даних виразу повинен співпадати з типом даних, що вказаний при оголошенні методу.

Варто зазначити, що оператор *return* повертає результат обчислення виразу в точку виклику методу.

Якщо тип виразу void, то повернення з методу виконується або після виконання останнього оператора тіла методу, або в результаті виконання оператора "return;"

Приклад оголошення методу, який повертає значення типу int – суму двох своїх параметрів типу int:

```
int sum(int a, int b){
    int x;
    x = a + b;
    return x;
}
```

Виклик методу для його виконання здійснюється у формі:

ім'я_методу(аргументи_методу);

Після імені методу вказуються дужки, в яких перераховуються аргументи (за необхідності).

Для наведеного вище прикладу при виклику методу, наприклад,

```
public static void test() {
    sum(8, 2); // Виклик методу sum з аргументами int a=8, int b=2
}
```

аргументи 8 і 2 передаються в метод, як значення відповідно a і b, і оператор виклику методу sum(8, 2) замінюється значенням, що повертається методом (для даного прикладу це значення дорівнює 10). Проте для даного прикладу результат виклику методу sum() нікуди не зберігається, тож його буде втрачено. Щоб виправити дану ситуацію можна замінити інструкцію виклику методу на наступну:

public static void test() {

int x=sum(8, 2); // Збереження результату виконання методу в цілочисельну змінну x

}

На відміну від мови С, в якій тип параметра, що задається при виклику, приводиться до типу параметра в оголошенні функції, тип параметра, що задається в Java повинен строго відповідати типу параметра в оголошенні методу, тому виклик методу sum(1.5, 8) призведе до помилки при компіляції програми.

Нестатичні методи в Java використовуються частіше, ніж статичні (з ключовим словом static у їх оголошенні) методи. Ці методи можуть належати будь-якому об'єкту (екземпляру класу), а не всьому класу. Дуже часто робота з нестатичними методами організовується таким чином, що вони можуть отримувати доступ і змінювати поля об’єкта.

Головний клас будь-якої програми на Java має обов’язково містити метод *main()*, який служить точкою входу в програму:

```
public static void main (String [] args) {
    // Тіло методу
}
```

Перевантаження методів

У мові Java в межах одного класу можна визначити два або більше методів, які мають однакове ім'я, але мають різну кількість параметрів або однакову кількість параметрів, але з різною комбінацією типів даних. Коли це має місце, методи називають перевантаженими, а про процес кажуть як про перевантаження методу (method overloading).

Коли метод викликається, то за кількістю параметрів і/або їх типам компілятор Java визначає, яку саме версію перевантаженого методу треба викликати.

Наприклад, метод

```
double sum (double a, double b) {
    double x;
    x=a+b;
    return x;
}
```

разом з оголошеним раніше методом *int sum (int a, int b) {}* складають пару перевантажених методів і при виклику sum(5, 8) буде викликаний перший метод, а при виклику sum(5.0, 8.0) буде викликаний другий метод.

Конструктор класу та об'єкти

В Java конструктор ініціалізує об'єкт при його створенні. По своїй суті конструктор є методом, тому його синтаксис схожий з синтаксисом звичайного методу. Однак, на відміну від останнього, конструктор має свої унікальні особливості, які відрізняють його від звичайного методу, а саме: не повертає ніяких значень (при оголошенні не вказується навіть ключове слово void), ім'я конструктора має бути таким же, як і ім'я класу, конструктор не може бути статичним.

Як правило, конструктор в Java може використовуватися для присвоєння початкового значення змінних екземпляра, що визначаються класом, або для виконання будь-яких інших процедур запуску, необхідних для створення повністю сформованого об'єкта.

Конструктори присутні у всіх класах. Якщо конструктор у класі відсутній, тобто явно не заданий, то компілятор Java автоматично створює конструктор за замовчуванням, який ініціалізує всі змінні членів класу нулем. Якщо у класі оголошений хоча б один конструктор, то конструктор за замовчуванням створюватись не буде.

Приклади оголошення конструкторів:

1. Конструктор без параметрів, а також конструктор за замовчуванням:

```
public class Student {
   Student() {
   }
}
```

або

```
public class Student {
   Student() {
      System.out.println("З'явився новий студент!");
      \\ Інші інструкції
   }
}
```

2. Конструктор з параметрами:

```
public class Student {
   private String name;
   Student(String studentName) {
      name=studentName;
      System.out.println("З'явився новий студент %s!", studentName);
      \\ Інші інструкції
   }
}
```

3. Перевантаження конструктора:

```
public class Student {
   private String name;
   private String group;
   Student(String studentName) {
      name=studentName;
      System.out.println("З'явився новий студент %s!", studentName);
      \\ Інші інструкції
   }
   Student(String studentName, String studentGroup) {
      name=studentName;
      group=studentGroup;
      System.out.println("У групі %s з'явився новий студент %s!", studentGroup, studentName);
      \\ Інші інструкції
   }
}
```

У Java для роботи з об'єктами використовується єдиний синтаксис. Об'єкт певного класу оголошується за допомогою створення посилання на нього. Загальна форма оголошення об'єкта класу без виділення пам'яті для нього (створення посилання) має наступний вигляд:

Ім'я_класу ім'я_об'єкта;

де *Ім'я_класу* – ім'я класу, об'єкт якого необхідно створити; *ім'я_об'єкта* – ім'я, яке є посиланням на об'єкт відповідного класу.

Вищенаведене оголошення говорить про те, що ідентифікатор *ім'я_об'єкта* є посиланням на об'єкт класу *Ім'я_класу*. Якщо необхідно ініціалізувати посилання об'єктом, то треба використати оператор *new* та викликати конструктор об'єктів як показано нижче:

ім'я_об'єкта = new Ім'я_класу();

Оператор *new* робить запит диспетчеру пам'яті для пошуку місця під об'єкт відповідного класу. Якщо не виділити пам'ять для посилання і звернутися до нього як до об'єкта класу, то виникне помилка.

Окрім вищезазначеного способу можна також оголошувати та ініціалізувати об'єкт в одній інструкції. У цьому випадку пам'ять під об'єкт виділяється одразу:

Ім'я_класу ім'я_об'єкта = new Ім'я_класу();

У Java для зберігання об'єктів або даних існує декілька сховищ:

1) **Регістри.** У цьому випадку дані зберігаються всередині процесора. У регістрах дані обробляються швидше за все, але кількість регістрів є дуже обмеженимою. Компілятор використовує регістри за необхідності. У Java немає безпосередніх команд, щоб зберігати всі дані тільки в регістрах.

2) **Стек.** Для програми стек розміщується в загальній оперативній пам'яті. Стек працює за принципом LIFO (Last-In-First-Out) – останній прийшов, перший вийшов. Така організація є зручною, коли потрібно виділяти пам'ять для локальних функцій (блоків, методів, змінних) різних рівнів вкладення. У стеці розміщуються тільки посилання на об'єкти. Самі об'єкти розміщуються в «купі». Посилання у стеці зсуваються вниз, якщо потрібно виділити пам'ять, і вгору, якщо пам'ять звільняється. Таким чином, за швидкодією, стек поступається тільки регістрам.

3) **«Купа» або динамічна пам'ять.** Це сховище загального призначення, яке розміщується в оперативній пам'яті. Тут зберігаються всі екземпляри класу (об'єкти) Java. «Купа» є більш гнучкою порівняно зі стеком. Це пояснюється тим, що компілятор не витрачає додаткових зусиль на визначення життєвого циклу об'єктів, які знаходяться в «купі». У програмі створення об'єкта відбувається з використанням оператора new. У

результаті роботи даного оператора виділяється пам’ять в «купі». Однак, виділення пам’яті в «купі» займає більше часу ніж в стеці.

4) **Статична пам’ять.** У програмах часто використовуються дані, які є незмінними протягом роботи програми – константи. Вони розміщуються в статичній пам’яті, яка формується на початку роботи програми і звільняється після її завершення.

5) **Зовнішнє сховище.** Зовнішнім сховищем можуть бути носії інформації, наприклад, жорсткий диск комп’ютера або Flash-накопичувач. Цей вид збереження даних дозволяє зберігати дані на носіях інформації, а потім відновлювати їх та переносити в оперативну пам’ять.

Порядок виконання роботи.

Основні завдання, які необхідно вирішити: навчитися працювати з класами та методами, ознайомитися та використати на практиці різні варіанти створення об’єктів за допомогою конструкторів, навчитися задавати характеристики та поведінку об’єктів.

Розв’язання: Для рішення перелічених завдань необхідно скористатися програмним пакетом IntelliJ IDEA.

Для роботи з характеристиками об’єктів зазвичай створюють, так звані, ‘гетери’ та ‘сетери’ – методи, які дозволяють відповідно отримувати значення характеристик об’єктів та задавати їх. Більш детально зображено у наступному прикладі:

```
public class Student {
    private String name;
    private String group;
    Student() {                                    // Конструктор
    }
```

```
    void setName(String studentName) {       // Задати ім'я студента
        name=studentName;
    }
    void setGroup (String studentGroup) {    // Задати групу студента
        group=studentGroup;
    }
    String getName() {                       // Отримати ім'я студента
        return name;
    }
    String getGroup() {                      // Отримати групу студента
        return group;
    }
}
class Main {                                 // Головний клас
    public static void main (String [] args) {
        Student stud1=new Student();
        stud1.setName('Олег');
        stud1.setGroup('ПГ-81');
        System.out.println('Студента групи %s звати %s', getGroup(), get-
Name());
    }
}
```

Основні особливості цих методів: 'гетери' не мають аргументів, при оголошенні мають тип даних, що буде повернено та обов'язково мають ключове слово *return*; 'сетери' не повертають ніяких значень (при оголошенні мають тип даних, що буде повернено – *void*), а також обов'язково мають аргументи.

Приклади створення класів, методів та конструкторів, а також варіанти їх оголошення та виклику наведено вище.

Звіт з практикуму має містити: стислі теоретичні відомості, умову завдання, код програми з необхідним описом, результати виконання, висновки по роботі.

Завдання для виконання.

Частина 1. Класи та конструктори. Кожен студент має обрати тему з табл. 1.7 (теми не мають повторюватись) та в її межах виконати наступні завдання:

а) Створіть клас (назва має бути в межах обраної теми), наприклад, ***Class***, який містить 3 поля (назва має бути в межах обраної теми; кожне поле має чітко висвітлювати конкретну властивість об'єкту): **va1**, **var2** та **var3**.

б) Створіть другий клас, наприклад, **Main**, в якому створіть три екземпляри класу ***Class***.

в) Виведіть на консоль значення їх полів.

г) Додати в клас **Main** два методи, які мають бути з параметрами та не можуть бути з порожнім тілом. Викликати один метод з іншого.

д) Додайте перевантажений метод. Викликати цей метод.

е) Додати конструктор в клас ***Class***, який приймає на вхід три параметра для ініціалізації полів класу **var1**, **var2** та **var3**.

є) Додати конструктор, який приймає на вхід два параметра для ініціалізації полів класу **var1** та **var2** початковими значеннями.

ж) Додати конструктор без параметрів, який ініціалізує поле класу **var3** початковим значенням, наприклад, **name="No name"**.

з) Викликати з конструктора з трьома параметрами конструктор з двома параметрами.

і) Додайте в клас методи **get** та **set** для правильної взаємодії з кожним полем екземпляру. Викликати кожен метод для кожного створеного об’єкта.

Частина 2. Створіть клас Wine, який в подальшому може бути застосовний при створенні електронного каталогу вин. Продумайте, які поля і методи знадобляться класу **Wine**, можна скористатися такими функціями:

1. Зберігання інформації про вид вина: назва, торгова марка, країна, дата розливу, примітка.

2. Доступ (установка і отримання значень) до інформації, що зберігається.

3. Розрахунок витримки вина (поточна дата дається як аргумент).

Окремо розробіть допоміжний клас, який демонструє працездатність класу **Wine**.

Ускладнене завдання (за вибором). Обрати довільну тему. Створити клас, наприклад, Animal і три класи, які його розширюють, наприклад, Dog, Cat, Horse. Клас Animal має містити мінімум два поля, наприклад, food, location та мінімум три методи, наприклад, makeNoise, eat, sleep. Метод makeNoise, наприклад, може виводити на консоль "Така-то тварина спить". Dog, Cat, Horse перевизначають (рос. переопределяют) однойменні методи makeNoise, eat, sleep. Додайте мінімум два поля в кожен підклас Dog, Cat, Horse, що характеризують тільки цих тварин. Створіть клас Ветеринар, в якому визначте метод void treatAnimal (Animal animal). Нехай цей метод роздруковує food і location тварини, яка прийшла на прийом. У методі main створіть масив типу Animal, в який запишіть тварин усіх наявних у вас типів. У циклі відправляйте їх на прийом до ветеринара, який має ставити випадковий діагноз.

Основні елементи, які мають бути в програмі за обраною темою: суперклас з двома полями та трьома методами (мінімум); три підкласи з двома додатковими (унікальними) полями та трьома методами (які перевизначають однойменні методи суперкласу) кожен; масив зі створеними об’єктами; додатковий клас зі своїми полями та методами, який веде взаємодію з об’єктами суперкласу та підкласів у циклі.

Контрольні запитання.

1. Що таке клас?
2. Які є модифікатори доступу?
3. Що таке метод?
4. Що таке конструктор?
5. Які відмінності конструктора від звичайного метода?
6. Що робить ключове слово new при створенні об’єкта?
7. Які типи сховищ можна використовувати в Java?

Комп’ютерний практикум №3

Введення даних з клавіатури

Мета роботи: розглянути основні методи введення даних з клавіатури, навчитися працювати з методами класу Scanner.

Теоретичні відомості

Потоки даних

У Java для опису роботи по введенню/виведенню використовується спеціальне поняття – потік даних (stream).

Потоки даних – це абстрактні впорядковані послідовності інструкцій або даних, яким відповідає певне джерело введення або, які мають певний пункт призначення (destination) виведення.

Відповідно, потоки діляться на вхідні – читають дані, і на вихідні – передають (записують) дані. У Java потоки введення/виведення реалізуються в межах ієрархії класів, які знаходяться в пакеті java.io. Класи введення/виведення Java виключають необхідність повного розуміння особливостей організації операційних систем на низькому рівні та надають доступ до системних ресурсів за допомогою методів роботи з файлами та інших інструментів.

Усі потоки введення/виведення поводяться однаково, незважаючи на відмінності в конкретних фізичних пристроях, з якими вони пов'язані. Одні і ті ж класи та методи введення/виведення можуть використовуватися з різнотипними пристроями. При цьому, абстракція потоку введення може

охоплювати різні типи введення: з файлу на диску, клавіатури або мережевого з'єднання.

У JAVA існує 3 типи потоків даних:

- символьні потоки (text-streams) – послідовності 16-бітових символів Unicode (рис. 3.1);
- байтові потоки (binary-streams) – містять 8-ми бітову інформацію (рис. 3.2). На відміну від символьних, байтові потоки введення/виведення реалізовані на найнижчому рівні;
- буферизовані потоки (buffered-streams) – читання з буфера або запис в буфер для підвищення ефективності.

Буфер – область пам’яті, що використовується для тимчасового зберігання даних при введенні/виведенні.

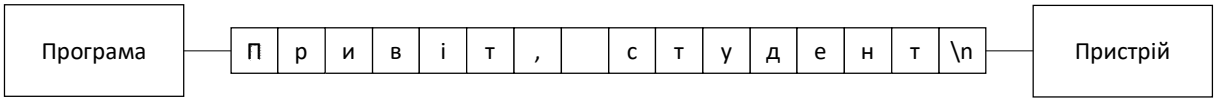

Рисунок 3.1 – Абстрактний приклад символьного потоку

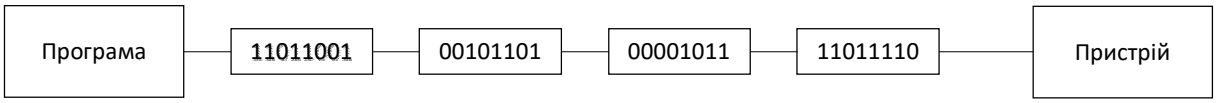

Рисунок 3.2 – Абстрактний приклад байтового потоку

Байтові та символьні потоки використовують небуферизоване введення/виведення, тобто кожен запит на введення/виведення обробляється безпосередньо базовою операційною системою. Це може призвести до зниження ефективності при роботі програми, оскільки кожен такий запит викликає доступ до диску, мережі або інших операцій, які є дорогими з точки зору системних ресурсів. Для зменшення подібних витрат системних

ресурсів, на платформі Java реалізовано буферизацію потоків введення/виведення.

У мові Java існує ряд класів, які забезпечують роботу байтових та символьних потоків введення/виведення, серед яких визначено чотири основних абстрактних класи для роботи з потоками (рис. 3.3).

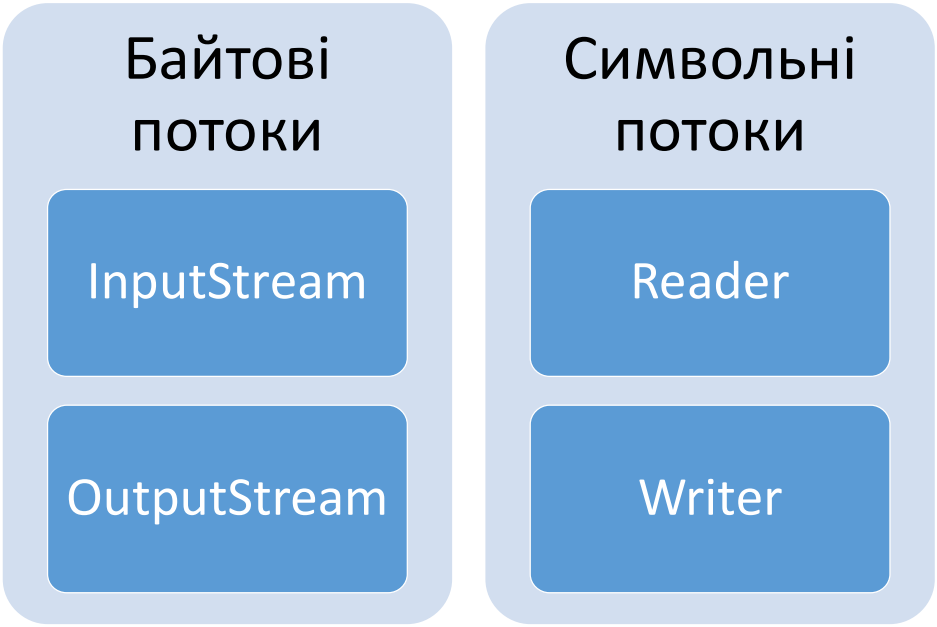

Рисунок 3.3 – Основні абстрактні класи для роботи з потоками

Класи, що успадковані від абстрактного класу **InputStream** і реалізують байтові потоки введення:

- **InputStream** – абстрактний клас, який описує потік введення, є базовим для всіх інших класів системи введення;
- **BufferedInputStream** – реалізує буферизований потік введення;
- **ByteArrayInputStream** – реалізує потік введення, який читає байти з масиву;

- **DataInputStream** – реалізує методи для зчитування даних примітивних типів;
- **FileInputStream** – реалізує потік введення, який читає дані з файлу;
- **FilterInputStream** – реалізація абстрактного класу **InputStream**;
- **ObjectInputStream** – реалізує потік введення об'єктів;
- **PipedInputStream** – реалізує концепцію ‘конвеєру’;
- **PushbackInputStream** – підтримує повернення одного байта назад в потік введення;
- **SequenceInputStream** – складається з двох або більше потоків введення, дані з яких читаються по черзі.

Класи, що успадковані від абстрактного класу **OutputStream** і реалізують байтові потоки виведення:

- **OutputStream** – абстрактний клас, який описує потік виведення. Усі інші класи системи виведення є підкласами класу **OutputStream**;
- **BufferedOutputStream** – реалізує буферизований потік виведення;
- **ByteArrayOutputStream** – реалізує потік виведення, який записує байти в масив;
- **DataOutputStream** – реалізує методи для зчитування даних примітивних типів;
- **FileOutputStream** – реалізує потік введення, який записує дані в файл;
- **FilterOutputStream** – реалізує абстрактний клас **OutputStream**;

- **ObjectOutputStream** – реалізує потік виведення об’єктів;
- **PipedOutputStream** – реалізує концепцію ‘конвеєру’;
- **PrintStream** – представляє собою потік виведення, що містить методи print() і println().

Варто зазначити, що програма може перетворити небуферизований потік в буферизований, використовуючи ідіому обтікання, де об'єкт небуферизованого потоку передається конструктору класу буферизованого потоку, наприклад:

```
BufferedReader inputStream = new BufferedReader(new FileReader("xanadu.txt"));
BufferedWriter outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Як зазначено вище, у бібліотеці класів *java.io.** визначені класи **PipedOutputStream** і **PipedInputStream**, за допомогою яких можна організувати конвеєрну передачу даних.

Конвеєрні потоки створюються парами, причому об’єкт одного з них, як параметр, передається в інший. Наприклад, можна спочатку створити вихідний конвеєрний потік класу **PipedOutputStream**, а потім доповнити його парою – вхідним конвеєрним потоком класу **PipedInputStream**:

```
PipedOutputStream pout = new PipedOutputStream();
PipedInputStream pin = new PipedInputStream(pout);
```

У класах **PipedOutputStream** і **PipedInputStream** визначені, відповідно, методи write() і read(), за допомогою яких можна працювати з примітивними типами даних. Якщо необхідно передавати через конвеєрні потоки дані реалізованих в Java типів або об'єкти довільних класів, що реалізують інтерфейс Serializable, треба створити на базі цих потоків потоки відповідних класів **DataOutputStream**, **DataInputStream**, **ObjectOutputStream** і **ObjectInputStream**.

Конвеєрні потоки зручно використовувати для синхронізованої передачі даних з одного потоку виконання в інший. При читанні даних з пус-

того вхідного конвеєрного потоку потік виконання буде блокований, доки там не з'являться дані. Ці дані обов’язково повинні бути записані у вихідний конвеєрний потік, прив'язаний до зазначеного вхідного конвеєрного потоку, іншим потоком виконання.

Стандартне введення/виведення

Термін ‘стандартне введення/виведення’ відноситься до концепції Unix (яка в деякій формі була відтворена в Windows і багатьох інших операційних системах) – це єдиний потік інформації, який використовується програмою. Все введення в програмі може відбуватися через стандартне введення, усе виведення в програмі може відбуватися через стандартне виведення, а всі повідомлення про помилки можуть надсилатися в стандартний потік помилок.

Клас **System** містить три поля – in, out і err. Вони використовуються для читання даних з **InputStream** і запису даних в **OutputStream**:

- змінна System.in – посилання на стандартний потік введення, якому відповідає клавіатура;
- змінна System.out – посилання на стандартний потік виведення, якому відповідає консоль;
- змінна System.err – посилання на стандартний потік виведення помилок. Цьому потоку відповідає консоль.

Отже, для отримання консольного введення в класі **System** визначено поле in. Однак безпосередньо через об'єкт System.in не дуже зручно працювати, тому, як правило, використовують клас **Scanner**, який, в свою чергу використовує System.in.

Клас Scanner

Функціональні можливості класу **Scanner** дуже схожі на можливості реального сканеру, а саме: зчитування даних з явно вказаного джерела.

Клас **Scanner** знаходиться в пакеті *java.util*, тому для роботи спочатку треба його імпортувати. Для імпорту будь-якого пакету необхідно на рядку, який передує ключовому слову class, прописати інструкцію, яка починається з ключового слова import. Для класу **Scanner** дана інструкція має наступний вигляд:

import java.util.Scanner;

Для створення самого об'єкта Scanner у його конструктор передається об'єкт System.in, після чого можна отримувати значення, які вводяться з клавіатури в консоль:

Scanner scan = new Scanner(System.in);

Клас **Scanner** має ще ряд методів, які дозволяють отримати введені користувачем значення (табл. 3.1).

Таблиця 3.1 – Методи класу **Scanner**

Метод	Значення, що повертає метод	Опис
1	2	3
close()	void	Закриває поточний сканер.
delimiter()	Pattern	Повертає шаблон, який використовує поточний сканер, щоб відповідати роздільникам.

Продовження таблиці 3.1

1	2	3
findInLine(Pattern pattern)	String	Спроба знайти наступне входження вказаного шаблону, ігноруючи розділь-ники.
findInLine(String pattern)	String	Спроба знайти наступне входження шаблону, побудованого із зазначеного рядка, ігноруючи роздільники.
findWithinHorizon (Pattern pattern, int horizon)	String	Спроба знайти наступне входження за-значеного шаблону.
findWithinHorizon (String pattern, int horizon)	String	Спроба знайти наступне входження шаблону, побудованого із зазначеного рядка, ігноруючи роздільники.
hasNext()	boolean	Повертає true, якщо цей сканер має ін-ший маркер на вході.
hasNext(Pattern pattern)	boolean	Повертає true, якщо наступний повний маркер відповідає вказаному шаблону.
hasNext(String pattern)	boolean	Повертає true, якщо наступний маркер відповідає шаблону, побудованому із зазначеного рядка.
hasNextBigDecimal()	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера мо-же бути інтерпретований як BigDecimal, використовуючи метод nextBigDecimal().

Продовження таблиці 3.1

1	2	3
hasNextBigInteger()	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера може бути інтерпретований як BigInteger у default radix за допомогою методу nextBigInteger().
hasNextBigInteger(int radix)	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера може бути інтерпретований як BigInteger у int radix за допомогою методу nextBigInteger().
hasNextBoolean()	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера може бути інтерпретований як логічне значення, використовуючи шаблон, нечутливий до регістру, створений із рядка "true | false".
hasNextByte()	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера може бути інтерпретований як значення байта у default radix за допомогою методу nextByte().
hasNextByte(int radix)	boolean	Повертає true, якщо наступний маркер на вході поточного сканера може бути інтерпретований як значення байта у int radix за допомогою методу nextByte().

Продовження таблиці 3.1

1	2	3
hasNextDouble()	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера може бути інтерпретований як double за допомогою методу nextDouble().
hasNextFloat()	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера може бути інтерпретований як float за допомогою методу nextFloat()
hasNextInt()	boolean	Повертає true, якщо наступний маркер на вході поточного сканера може бути інтерпретований як значення int у default radix за допомогою методу nextInt().
hasNextInt(int radix)	boolean	Повертає true, якщо наступний маркер на вході поточного сканера може бути інтерпретований як значення int у int radix за допомогою методу nextInt().
hasNextLine()	boolean	Повертає true, якщо на вході поточного сканера є інший рядок.
hasNextLong()	boolean	Повертає true, якщо наступний маркер на вході поточного сканера може бути інтерпретований як long у default radix за допомогою методу nextLong().

Продовження таблиці 3.1

1	2	3
hasNextLong(int radix)	boolean	Повертає true, якщо наступний маркер на вході поточного сканера може бути інтерпретований як long у int radix за допомогою методу nextLong().
hasNextShort()	boolean	Повертає true, якщо наступний маркер у вхідних даних поточного сканера може бути інтерпретований як short у default radix за допомогою методу nextShort().
hasNextShort(int radix)	boolean	Повертає true, якщо наступний маркер на вході поточного сканера може бути інтерпретований як short у int radix за допомогою методу nextShort().
ioException()	IOExceptio n	Повертає останній викликаний поточним сканером виняток IOException.
locale()	Locale	Повертає локаль поточного сканера.
match()	MatchResul t	Повертає результат збігу останньої операції сканування, виконаної поточного сканером.
next()	String	Знаходить і повертає наступний повний маркер із поточного сканера.
next(Pattern pattern)	String	Повертає наступний маркер, якщо він відповідає вказаному шаблону.

Продовження таблиці 3.1

1	2	3
next(String pattern)	String	Повертає наступний маркер, якщо він відповідає шаблону, побудованому із зазначеного рядка.
nextBigDecimal()	BigDecimal	Сканує наступний маркер вводу як BigDecimal.
nextBigInteger()	BigInteger	Сканує наступний маркер вводу як BigInteger.
nextBigInteger(int radix)	BigInteger	Сканує наступний маркер вводу як BigInteger.
nextBoolean()	boolean	Сканує наступний маркер вхідних даних у логічне значення і повертає це значення.
nextByte()	byte	Сканує наступний маркер вводу як byte.
nextByte(int radix)	byte	Сканує наступний маркер вводу як byte.
nextDouble()	double	Сканує наступний маркер вводу як double.
nextFloat()	float	Сканує наступний маркер вводу як float.
nextInt()	int	Сканує наступний маркер вводу як int.
nextInt(int radix)	int	Сканує наступний маркер вводу як int.
nextLong()	long	Сканує наступний маркер вводу як long.
nextLong(int radix)	long	Сканує наступний маркер вводу як long.

Продовження таблиці 3.1

1	2	3
nextShort()	short	Сканує наступний маркер вводу як short.
nextShort(int radix)	short	Сканує наступний маркер вводу як short.
radix()	int	Повертає default radix поточного сканера.
remove()	void	Операція видалення не підтримується поточною реалізацією ітератора.
reset()	Scanner	Скидає поточний сканер.
skip(Pattern pattern)	Scanner	Пропускає введення, яке відповідає вказаному шаблону, ігноруючи роздільники.
skip(String pattern)	Scanner	Пропускає введення, яке відповідає шаблону, побудованому із зазначеного рядка.
toString()	String	Повертає рядкове представлення поточного сканера.
useDelimiter(Pattern pattern)	Scanner	Замінює шаблон розмежування представлення поточного сканера на вказаний.
useDelimiter(String pattern)	Scanner	Замінює шаблон розмежування поточного сканера на зразок, побудований із зазначеного рядка.
useLocale(Locale locale)	Scanner	Замінює локаль поточного сканера на вказану мову.

Продовження таблиці 3.1

1	2	3
useRadix(int radix)	Scanner	Замінює default radix поточного сканера на int radix.
nextLine()	byte	Сканує наступний маркер вводу як String.

У позиційній системі числення radix або основа – це кількість унікальних цифр, включаючи нуль, що використовуються для представлення чисел. Наприклад, для десяткової системи radix (базове число) дорівнює десяти, оскільки він використовує десять цифр від 0 до 9.

Приклад використання класу **Scanner** та його методів наведено нижче:

```
Scanner scan = new Scanner(System.in);
System.out.print("Input name: ");
String name = scan.nextLine();
System.out.print("Input age: ");
int age = scan.nextInt();
System.out.print("Input height: ");
float height = scan.nextFloat();
System.out.printf("Name: %s  Age: %d  Height: %.2f \n", name, age, height);
scan.close();
```

Незважаючи на зручність класу **Scanner**, якщо треба просто зчитувати рядки без їх аналізу, то краще використовувати **BufferedReader**, так як він працює швидше, але вибір конкретного класу введення даних залежить від специфіки завдання:

```
InputStream a = System.in;
```

```
InputStreamReader b = new InputStreamReader(a);
BufferedReader reader = new BufferedReader(b);
```

або

```
BufferedReader reader = new BufferedReader (new InputStreamReader(System.in));
```

Окрім введення даних з клавіатури клас **Scanner** дозволяє працювати з іншими джерелами даних, наприклад, статичними рядками:

```
Scanner scanner = new Scanner("Як умру, то поховайте\n" +
"Мене на могилі,\n"
+ "Серед степу широкого,\n"
+ "На Вкраїні милій,..");
System.out.println(scanner.nextLine());
System.out.println(scanner.nextLine());
System.out.println(scanner.nextLine());
System.out.println(scanner.nextLine());
```

Робота з файлами та каталогами

Клас **File** пакета *java.io* використовується для управління інформацією про файли та каталоги. На рівні операційної системи файли і каталоги мають істотні відмінності, але в Java вони описуються одним і тим же класом. Проте, клас **File**, всупереч назві, є шаблоном не файлу, а шляху до файлу або підкаталогу. Це значить, що створення екземпляра класу **File** ніяк не пов'язане зі створенням самого файлу (для роботи з файлами служать потоки введення/виведення). По суті, клас **File** можна вважати спеціалізованим варіантом класу **String**, розрахованим на роботу зі шляхами до файлів/каталогів, який допускає створення спадкоємців (підкласів).

Файлова система – це засіб для організації зберігання файлів на будь-якому носії у відповідності до суворої ієрархії елементів (дисків, папок, файлів, тощо).

Шлях – це набір символів, що показує розташування файлу або каталогу в ієрархії файлової системи.

В Unix-системах (наприклад, Linux чи Mac) використовується один загальний кореневий каталог, який позначається косою рискою (/), а диски позначаються так само, як і папки. При цьому елементи файлової системи поділяються косою рискою (/), наприклад, /files/images/picture.jpg. У цьому записі адресації picture.jpg – ім'я файлу, а /files/images/ – шлях до нього.

На відміну від Unix, у Windows немає єдиного кореневого каталогу. Натомість весь простір у Windows поділений на, так звані, диски, кожен з яких є кореневим каталогом. Диск позначається однією з букв алфавіту, а також символом двокрапки на кінці цієї букви, а елементи файлової системи поділяються символом зворотної косої риски, наприклад, C:\ProgramFiles.

Шлях може бути абсолютним або відносним. Повний або абсолютний шлях – це шлях, який вказує на одне і те ж місце в файловій системі, незалежно від поточного робочого каталогу або інших обставин. Повний шлях завжди починається з кореневого каталогу. Наприклад, для Windows абсолютний шлях виглядатиме C:\ProgramFiles\Java\bin, а для Unix – /files/images/wallpapers/cats.

Відносний шлях – це шлях, який визначається відносно поточного робочого каталогу користувача або активних додатків. Наприклад, для Windows відносний шлях виглядатиме \Java\bin, а для Unix – images/wallpapers/cats.

Для того щоб вказати шлях від файлу, який стоїть нижче по ієрархії файлової системи, до файлу, який стоїть вище, потрібно використовувати

‘вихід з каталогу’ – дві послідовні точки (..). Для деяких Unix-систем необхідно також використовувати шлях ‘поточний каталог’, який має позначення точки перед косою рискою, наприклад, **./**picture.jpg.

Наприклад, для зображеної на рис. 3.4 ієрархії каталогів можна використати наступні відносні шляхи для переходу до інших каталогів, враховуючи, що початковим каталогом є jdk-13.0.1:

- **..**\jdk-14 – відповідає абсолютному шляху C:\ProgramFiles\Java\jdk-14;
- **..** – відповідає абсолютному шляху C:\ProgramFiles\Java;
- **..\..** – відповідає абсолютному шляху C:\ProgramFiles;
- **..\..**\Softland – відповідає абсолютному шляху C:\ProgramFiles\Softland;
- **.** – відповідає абсолютному шляху C:\ProgramFiles\Java\jdk-13.0.1, тобто поточному каталогу;
- **.\..**\jdk-14 – відповідає абсолютному шляху C:\ProgramFiles\Java\jdk-14;
- **..\..\.**\Softland**\..**\Java**\.**\jdk-14 – відповідає абсолютному шляху C:\ProgramFiles\Java\jdk-14.

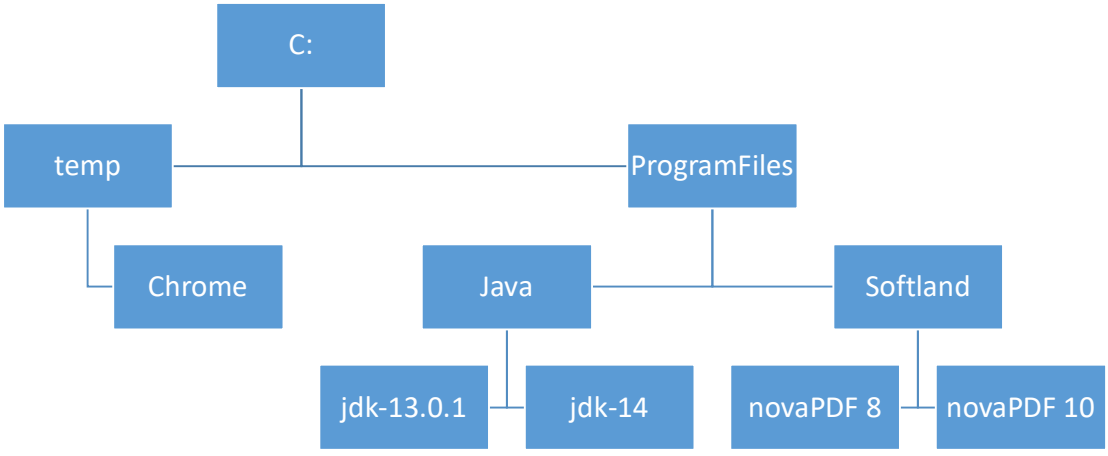

Рисунок 3.4 – Приклад ієрархії каталогів Windows

Отже, у класі **File** реалізовано декілька конструкторів, кожен з яких дозволяє формувати шлях до файлу або каталогу різними способами.

Найбільш поширеним є конструктор, який отримує рядок типу **String**. У цьому рядку задається повний (абсолютний) або скорочений (відносний) шлях до файлу або каталогу, який розглядається (створюється, визначається, тощо):

```
File f = new File("output.txt");
String path = f.getPath();
System.out.print(path);
```

У результаті виконання в консоль буде виведено *output.txt*.

Також існують конструктори, формують ім'я файлу з декількох частин:

```
File f = new File("G:\\", "Files");
String path = f.getPath();
System.out.println("path = " + path);
```

або

```
File f1 = new File("G:\\ProgramFiles\\Java\\Project22");
File f2 = new File(f1,"output.txt");
String path1 = f1.getPath();
String path2 = f2.getPath();
System.out.println("path1 = " + path1);
System.out.println("path2 = " + path2);
```

У результаті виконання в консоль буде виведено відповідно:

path = G:\Files

та

path1 = G:\ProgramFiles\Java\Project22

path2 = G:\ProgramFiles\Java\Project22\output.txt

Клас **File** має ряд методів, які дозволяють керувати файлами та каталогами. Основні з них наведено в табл. 3.2.

Таблиця 3.2 – Методи класу **File**

Метод	Величина, яку повертає	Опис
1	2	3
createNewFile()	boolean	Створює новий файл за шляхом, який переданий в конструктор. У разі вдалого створення повертає true, інакше false. Створює файл тільки у тому випадку, якщо файл з таким іменем не існує.
delete()	boolean	Видаляє каталог або файл за шляхом, який переданий в конструктор. При вдалому видаленні повертає true.
exists()	boolean	Перевіряє, чи існує за вказаним у конструкторі шляхом файл або каталог. Якщо файл або каталог існує, то повертає true, інакше повертає false.
getAbsolutePath()	String	Повертає абсолютний шлях за шляхом переданого в конструктор об'єкта.
getName()	String	Повертає коротке ім'я файлу або каталогу.

Продовження таблиці 3.2

1	2	3
getParent()	String	Повертає ім'я батьківського каталогу.
isDirectory()	boolean	Повертає значення true, якщо за вказаним шляхом розташовується каталог.
isFile()	boolean	Повертає значення true, якщо за вказаним шляхом знаходиться файл.
isHidden()	boolean	Повертає значення true, якщо каталог чи файл є прихованими.
length()	long	Повертає розмір файлу в байтах.
lastModified()	long	Повертає час останньої зміни файлу або каталогу. Значення виводиться в мілісекундах. Для переведення в дату необхідно скористатися класом **Date** (Date dt = new Date(f.lastModified()).
list()	String[]	Повертає масив файлів і підкаталогів, які знаходяться в певному каталозі.
listFiles()	File[]	Повертає масив файлів, які знаходяться в певному каталозі.
mkdir()	boolean	Створює новий каталог і при вдалому створенні повертає значення true.
renameTo(File dest)	boolean	Перейменовує файл або каталог.

Порядок виконання роботи.

Основні завдання, які необхідно вирішити: навчитися працювати з потоками введення даних з клавіатури, ознайомитися та використати на практиці при виконанні завдання для самостійної роботи методи класів **Scanner** та **File**.

Розв'язання: Для рішення перелічених завдань необхідно скористатися програмним пакетом IntelliJ IDEA.

Для роботи з класами **Scanner** та **File** в першу чергу необхідно підключити необхідні бібліотеки, відповідно *import java.util.Scanner* та *import java.io.File*. Для роботи з деякими методами класу (наприклад, createNewFile()) **File** додатково треба підключити бібліотеку *import java.io.IOException*, яка дозволяє відслідковувати виняткові ситуації.

Приклади роботи з методами класу **Scanner** наведено вище.

Переважна більшість методів з класу **File** мають схожий принцип використання з різницею лише в типі даних, який вони повертають. Наприклад, результатом виконання наступного коду

```
public static void main(String[] args) throws IOException {
File f1 = new File("G:\\Test");
File f2 = new File("G:\\Test\\output.txt");
String path1 = f2.getAbsolutePath();
System.out.println(path1);
f1.mkdir();
f2.createNewFile();
System.out.println(f2.getName());}
```

буде створення каталогу за адресою G:\Test, створення файлу за адресою G:\Test\output.txt, а також у консоль буде виведено наступне:

```
G:\Test\output.txt
output.txt
```

Як видно з результату та відповідно з табл. 3.2, можна сказати, що використані методи виконали свої функції. Також варто зазначити, що виконання методу createNewFile() призведе до помилки у тому випадку, коли хоча б один з каталогів або підкаталогів у шляху до файлу не існує. Для рішення цієї проблеми у наведеному прикладі перед створенням файлу output.txt відбувається створення каталогу Test за допомогою методу mkdir(). Також можуть виникати помилки при роботі з каталогами та файлами на системному диску (диск C: за замовчуванням).

Звіт з практикуму має містити: стислі теоретичні відомості, умову завдання, код програми з необхідним описом, рисунок з ієрархією каталогів за обраною темою, результати виконання, висновки по роботі.

Завдання для виконання.

Частина 1. Для завдань з практичної роботи №1 та практичної роботи №2 коректно реалізувати введення даних з клавіатури за допомогою класів **Scanner** та **BufferedReader**.

Частина 2. Обрати тему з табл. 3.3 та відповідно до обраної теми (цифрові позначення каталогів замінити словами чи словосполученнями за обраною темою) реалізувати ієрархію каталогів, зображену на рис. 3.5. Коректно реалізувати введення даних з клавіатури за допомогою класів **Scanner** та **BufferedReader**. Виконати наступні дії, використовуючи тільки відносні шляхи (якщо не вказано інше):

а) Обрати довільний початковий каталог та створити об’єкт класу **File** з використанням абсолютного шляху.

б) З поточного каталогу виконати послідовність переходів 1.2-2.1.2-2.2-1-2.2.2-2.1.2.1-2.1.1.1 та створити 3 текстових файли за обраною темою.

в) З поточного каталогу (2.1.1.1) виконати послідовність переходів 2.1.1.1-2.1.2-2-1.1-1.3 та створити 2 текстових файли за обраною темою.

Таблиця 3.3 – Теми для завдань

Тварини	Книги	Космос	Одяг	Іграшки
Магазин	Освіта	Друзі	Продукти	Дерева
Кіно	Фільми	Телефони	Автомобілі	Відпочинок
Квіти	Косметика	Ігри	Меблі	Подарунки
Мотоцикли	Час	Медицина	Серіали	Програмування
Техніка	Хобі	Спорт	Промисловість	Гаджети

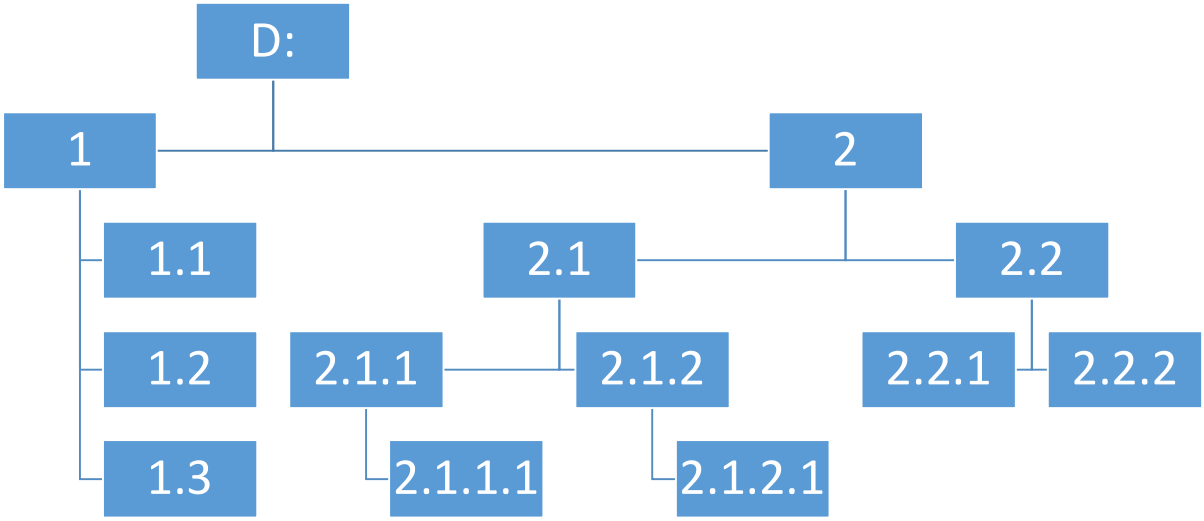

Рисунок 3.5 – Ієрархія каталогів для виконання завдання

г) З поточного каталогу (1.3) виконати послідовність переходів 1-2.1-D:-1.1-1.1-2.1.1-2.2.2 та створити 2 текстових файли за обраною темою.

д) З поточного каталогу (2.2.2) виконати послідовність переходів 2.2.2-2.2.2-D:-1.2-2.1-2.2.1 та створити 2 текстових файли за обраною темою.

е) З поточного каталогу (2.2.1) виконати послідовність переходів 1-2.2.1-1.3-2.1.1.1-2-2-2-1 та створити 4 текстових файли за обраною темою.

є) Використовуючи методи класу **File**, видалити по одному зі створених у попередніх пунктах текстовому файлу.

ж) Використовуючи методи класу **File**, визначити чи існує каталог з пункту а).

з) Обрати по одному файлу з пунктів б)-е) та використати до них по одному методу з табл. 3.2. Використаний для одного файлу метод повторно використовуватися не може.

Ускладнене завдання (за вибором). Створити гру ‘Морський бій’ для двох гравців за класичними правилами, а саме: кожен з гравців під час свого ходу може виконати постріл за певними координатами (x, y); якщо під час пострілу гравець влучив, то він може виконати додатковий постріл, у іншому випадку хід переходить другому гравцю; перемагає той гравець, котрий потопить усі ворожі кораблі першим. Ігрове поле має розмір 10x10, кількість кораблів: чотири 1-палубних; три 2-палубних; два 3-палубних; один 4-палубний. Розміщення кожного окремого корабля має бути або горизонтальним, або вертикальним. Розміщення кораблів по діагоналі заборонено. Приклад ігрового поля зображено на рис. 3.6.

Основні елементи, які мають бути в програмі: розстановка кораблів з підтвердженням вибору, захист від підглядування розстановки, перевірка введення координат при пострілах, виведення для кожного гравця ігрового поля з попередніми пострілами, виведення результатів гри з ігровими полями обох гравців, розташованими поруч (горизонтально на одному рівні) один з одним для найкращого візуального сприйняття результатів.

	Р	Е	С	П	У	Б	Л	І	К	А
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										

Рисунок 3.6 – Приклад ігрового поля

Контрольні запитання.

1. Що таке потік? Які бувають види потоків?

2. Які є основні абстрактні класи для роботи з потоками?

3. Що таке Scanner? Як створити та використовувати екземпляр класу Scanner?

4. За що відповідає клас **File**?

5. Як створити файл та каталог?

6. Що таке абсолютний шлях?

7. Що таке відносний шлях?

Практична робота №4

Використання розгалуження

Мета роботи: розглянути та порівняти можливості основних конструкцій розгалуження.

Теоретичні відомості

Розгалуження і вибір

Структури розгалуження і вибору – це структури, які беруть одне або більше логічних значень, що оцінюються програмою і, в залежності від істинності чи хибності цього логічного значення виконуються різні частини коду.

Мова програмування Java пропонує декілька типів операторів розгалуження, які представлені в табл. 4.1.

Таблиця 4.1 – оператори розгалуження

if	Спрощений варіант конструкції *if-else* без альтернативної гілки.
if-else	Оператор *if* може супроводжуватися додатковим оператором *else*, який виконується при неправдивому логічному виразі (хибній умові).
switch-case	Оператор розгалуження (вибору) *switch* дає можливість перевірити змінну на рівність з одним зі списку значень.

Продовження таблиці 4.1

?:	Тернарний умовний оператор. Є заміною однорівневої конструкції *if-else*.

Загальну схему розгалуження зображено на рис. 4.1.

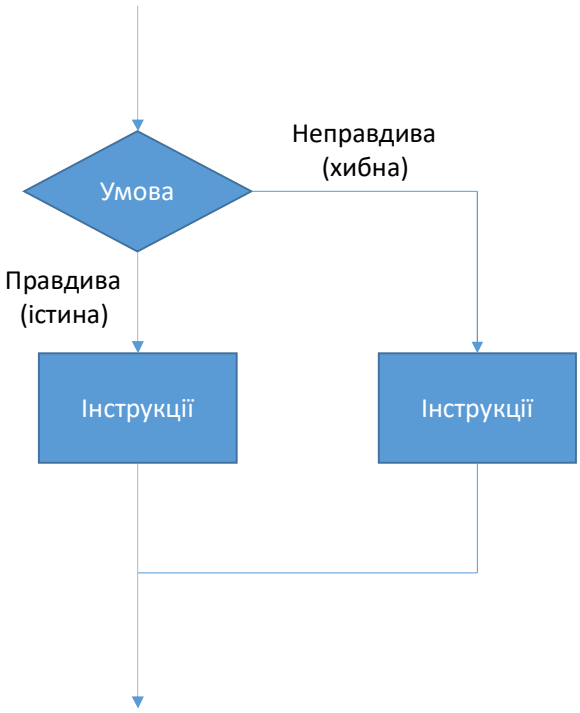

Рисунок 4.1 – Загальна схема розгалуження

Логічні операції

Логічні операції виконуються за допомогою логічних операторів. Основні логічні операції (в програмуванні та математиці) можна застосовувати до аргументів (операндів), а також складати більш складні вирази, подібно до арифметичних дій над числами. Наприклад вираз:

(a | b) | (c >7) &! (false) ^ (d == 45)

являє собою складний логічний вираз з чотирма операндами: (a | b), де a і b – змінні типу boolean; (c >7); (false); (d == 45). У свою чергу, простий логічний вираз (a | b) також складається з двох аргументів-операндів.

Логічний операнд – це вираз, результатом якого є тип даних **boolean** або **Boolean**, наприклад:

- (2 <1) – логічний операнд, його значення дорівнює false;
- true – логічний операнд, значення якого true;
- **boolean** a – логічний операнд, як і **Boolean** a;
- **int** a = 2 – не є логічним операндом, це змінна типу **int**;
- **String** a = "true" також не є логічним операндом, а рядок, текстове значення якого – "true".

У Java присутні три логічні операції:

Логічне заперечення (NOT) або інверсія. У Java позначається символом ‘!’ перед операндом. Ніколи не застосовується до групи операндів, а лише до одного операнду.

Логічне ‘і’ (AND) або кон’юнкція. Позначається символом ‘&&’ між двома операндами, до яких застосовується.

Логічне ‘або’ (OR) або диз’юнкція. У Java позначається символом ‘||’ між двома операндами, до яких застосовується.

Припустимо, що логічна змінна A має значення true, а змінна B – false. Логічні оператори наведено в табл. 4.2.

Таблиця 4.2 – Логічні оператори

Оператор	Опис	Приклад
&&	Якщо обидва операнда одночасно не дорівнюють нулю (false), то в результаті буде одиниця (true).	(A && B) – значення false

Продовження таблиці 4.2

||	Якщо будь-який з двох операндів не дорівнює нулю (false), то в результаті буде одиниця (true).	(A || B) – значення true
!	Логічний оператор «НЕ». Використання змінює логічне значення операнда. Якщо умова має значення true, то оператор логічного «НЕ» змінюватиме значення на false.	!(A && B) — значення true

Приклад використання логічних операторів у програмі:

boolean a = true;

boolean b = false;

System.out.println("a && b = " + (a&&b));

System.out.println("a || b = " + (a||b));

System.out.println("!(a && b) = " + !(a && b));

У результаті в консоль буде виведено наступне:

a && b = false

a || b = true

!(a && b) = true

Окрім логічних операторів у Java присутні декілька побітових операторів, які можна застосовувати для цілочисельних типів: **int**, **long**, **short**, **char** і **byte**. У Java побітовий оператор працює над бітами і виконує операцію біт за бітом. Також слід зазначити, що деякі з побітових операторів працюють і з логічними значеннями типу **boolean** або **Boolean**. Перелік побітових операторів наведено у табл. 4.3.

Наприклад, є a = 60, а b = 13 або у двійковій формі a = 0011 1100 та b = 0000 1101.

Таблиця 4.3 – Побітові оператори

Оператор	Опис	Приклад
&	Бінарний оператор AND копіює біт в результат, якщо цей біт присутній одночасно в обох операндах.	(A & B) дасть 12 (0000 1100)
|	Бінарний оператор OR копіює біт в результат, якщо цей біт присутній в будь-якому з операндів.	(A | B) дасть 61 (0011 1101)
^	Бінарний оператор XOR копіює біт в результат, якщо цей біт присутній лише в одному операнді.	(A ^ B) дасть 49 (0011 0001)
~	Бінарний оператор доповнення, який має ефект «відображення» біт.	(~ A) дасть -61, (1100 0011)
<<	Бінарний оператор зсуву вліво. Значення лівих операндів переміщається вліво на кількість біт, заданих правим операндом.	A << 2 дасть 240 (1111 0000)
>>	Бінарний оператор зсуву вправо. Значення лівих операндів переміщається вправо на кількість біт, заданих правим операндом.	A >> 2 дасть 15 (1111)
>>>	Нульовий оператор зсуву вправо. Значення лівих операндів переміщається вправо на кількість біт, заданих правим операндом, а зсунуті значення заповнюються нулями.	A >>> 2 дасть 15 (0000 1111)

Також під час роботи з логічними операціями дуже часто доводиться використовувати оператори присвоєння (табл. 4.4) та оператори порівняння (табл. 4.5).

Таблиця 4.4 – Оператори присвоєння

Оператор	Опис	Приклад
=	Простий оператор присвоєння, присвоює значення операндів з правого боку до лівого операнду.	A = B + C
+=	Скорочений оператор присвоєння ‘додавання’.	A += B еквівалентно A = A + B
-=	Скорочений оператор присвоєння ‘віднімання’.	A -= B, еквівалентно A = A - B
*=	Скорочений оператор присвоєння ‘множення’.	A * = B еквівалентно A = A * B
/=	Скорочений оператор присвоєння ‘ділення’.	A /= B еквівалентно A = A / B
%=	Скорочений оператор присвоєння ‘модуль’ (залишок від ділення).	A %= B, еквівалентно A = A % B
<<=	Скорочений оператор присвоєння ‘зсув вліво’	A << = 2, еквівалентно A = A << 2
>>=	Скорочений оператор присвоєння ‘зсув вправо’	A >>= 2, еквівалентно A = A >> 2
&=	Скорочений оператор присвоєння побітового I (AND)	A &= 2, еквівалентно A = A & 2
^=	Скорочений оператор присвоєння побітового виключного АБО (XOR)	A ^= 2, еквівалентно A = A ^ 2
|=	Скорочений оператор присвоєння побітового АБО (OR)	A |= 2, еквівалентно A = A | 2

Для наступної таблиці припустимо, що змінна A дорівнює 10, а змінна B дорівнює 20.

Таблиця 4.4 – Оператори порівняння

Оператор	Опис	Приклад
==	Перевіряє на рівність значення двох операндів, якщо так, то умова стає істинною	(A == B) – false
!=	Перевіряє на рівність значення двох операндів, якщо значення не рівні, то умова стає істинною	(A != B) – true
>	Перевіряє чи значення лівого операнда є більшим за значення правого операнда, якщо так, то умова стає істинною	(A > B) – false
<	Перевіряє чи значення лівого операнда є меншим за значення правого операнда, якщо так, то умова стає істинною	(A < B) – true
>=	Перевіряє чи значення лівого операнда є більшим або рівним значенню правого операнда, якщо так, то умова стає істинною	(A >= B) – false
<=	Перевіряє чи значення лівого операнда є меншим або рівним значенню правого операнда, якщо так, то умова стає істинною	(A <= B) – true

Якщо оператори зустрічаються в одному виразі, то як і в математиці, у програмуванні в операторів є певний порядок виконання. Унарні оператори мають переваги над бінарними, а множення (навіть логічне) над сумою. Нижче наведені пріоритети деяких логічних та бінарних операторів:

1. !
2. &
3. ^
4. |
5. &&
6. ||

Конструкція умовного переходу if-else

Конструкція умовного переходу дозволяє організувати розгалуження процесу виконання в програмі. Оператор умовного переходу може також містити інші оператори. Це можуть бути оператори, які реалізують послідовне виконання, циклічний процес а також умовний перехід.

Оператор умовного переходу має дві форми представлення:

- повна форма *if-else*;
- скорочена форма *if*.

Синтаксис конструкції *if-else* в Java наступний:

```
if (умова){
   // Виконується, якщо умова (логічний вираз) істинна
   Інструкція 1;
   Інструкція 2;
...
} else {
```

```
   // Виконується, якщо умова (логічний вираз) неправдива (хибна)
   Інструкція 1;
   Інструкція 2;
...
}
```

Якщо логічний операнд (логічний вираз або умова) істинний, то буде виконано блок коду *if*, в іншому випадку буде виконано блок коду *else*. Якщо тіло складається з одного виразу, фігурні дужки можна не використовувати:

```
if (умова) Інструкція;
else Інструкція;
```

У залежності від умови конструкція *if-else* може мати два вигляди:

а) умови незалежні або належать різним рівням ієрархії:

```
if (умова1){
   Інструкція 1;
...
} else {
   if (умова2){
   Інструкція 1;
   } else {
      Інструкція 1;
      }
   Інструкція 1;
...
}
```

б) умови залежні або належать одному рівню ієрархії:

```
if (умова1){
   Інструкція 1;
```

```
...
} else if (умова2){
   Інструкція 1;
...
}
```

Приклад для першої конструкції, а також скороченої форми запису наведено нижче:

```
if (i == 10) {
   if(j < 20) a = b;
   if(k > 100) c = d;
   else a = c;  // else відноситься до if(k > 100)
} else  a = d; // else відноситься до if(i == 10)
```

Для другої:

```
int month = 3; // березень
String season; // пора року
if(month == 1 || month == 2 || month == 12)
   season = "Зима";
else if (month == 3 || month == 4 || month == 5)
   season = "Весна";
else if (month == 6 || month == 7 || month == 8)
   season = "Літо";
else if (month == 9 || month == 10 || month == 11)
   season = "Осінь";
else
   season = "Немає?";
System.out.println("Коти під вікнами кричать, коли на вулиці " + season);
```

Тернарний оператор

Дуже часто однорівнева конструкція *if-else* замінюється іншою – тернарним оператором *?:*. Тернарний оператор використовує три операнда і записується в формі:

логічний_вираз? вираз1: вираз2;

Якщо логічний_вираз істинний, тобто повертає true, то виконується вираз1. Якщо логічний_вираз повертає false, то виконується вираз2.

Наприклад, потрібно обчислити, яке з двох чисел є більшим і записати результат в третю змінну:

```
int largerNum;
int lowNum = 9;
int highNum = 27;
if (lowNum <highNum) {   // якщо перше число менше другого
   largerNum = highNum;
} else     largerNum = lowNum;
```

При використанні тернарного оператора код буде наступним:

```
int lowNum = 9, highNum = 27;
int largerNum = lowNum <highNum? highNum: lowNum;
```

На практиці тернарний оператор використовується не часто через невелику кількість однорівневих конструкцій *if-else*.

Конструкція switch-case

Іноді, коли необхідно виконати вибір із багатьох варіантів, конструкція *if-else* може виявитися незручною. У таких випадках її можна замінити конструкцією *switch-case* (рис. 4.2).

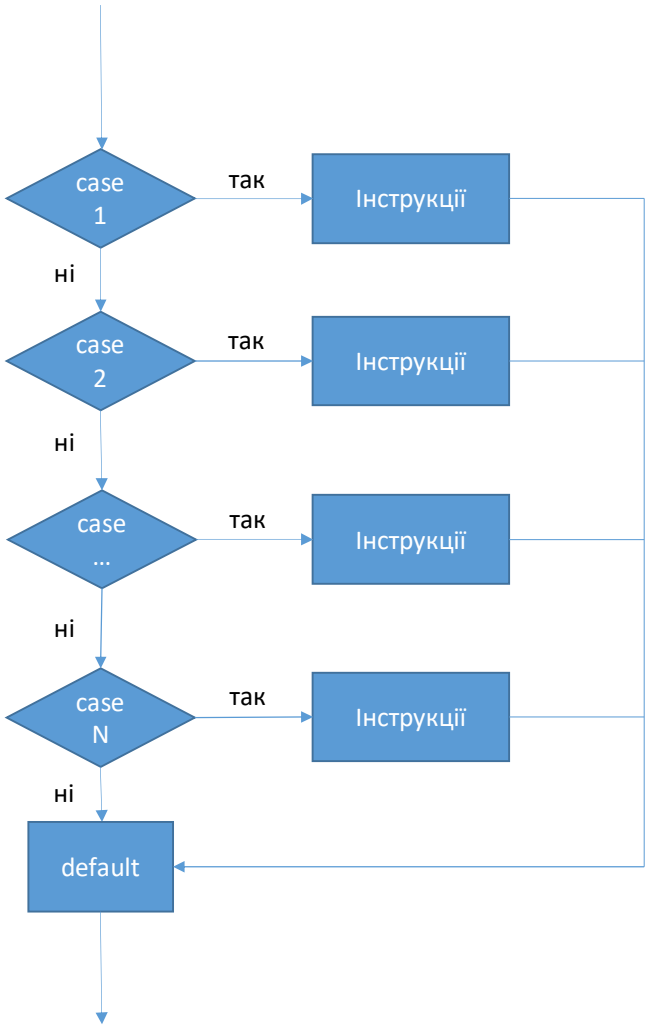

Рисунок 4.2 – Загальна схема конструкції switch-case

Конструкцію *switch-case* часто називають оператором вибору. Вибір здійснюється залежно від значення виразу або змінної. Загальна форма конструкції виглядає наступним чином:

switch (вираз) {

case (вираз_для_порівняння_1): Інструкція 1;…; Інструкція N; break;

case (вираз_для_порівняння_2): Інструкція 1;…; Інструкція N; break;

case (вираз_для_порівняння_...): Інструкція 1;…; Інструкція N; break;

case (вираз_для_порівняння_N): Інструкція 1;…; Інструкція N; break;

default: Інструкція 1;…; Інструкція N; break;

}

Параметр ‘вираз’ – це вираз, у результаті обчислення якого виходить як правило ціле число. У якості виразу можна також використовувати прості типи **byte**, **short**, **char**, **int**, спеціальні класи, які є обгорткою для примітивних типів (**String**, **Character**, **Byte**, **Short**, **Integer**), а також тип **enum**. Команда switch порівнює результат виразу з кожним наступним значенням ‘вираз_для_порівняння’. Якщо буде виявлено збіг, то виконуються інструкції, які прописані після двокрапки за оператором *case*. Якщо збігів не буде, то виповнюється інструкції після ключового слова *default*, який не є обов’язковим. У випадку відсутності *default*, якщо немає збігів, програма не виконує ніяких дій.

Кожна секція *case* зазвичай закінчується командою *break*, яка передає управління в кінець команди *switch*.

Розглянемо найпростіший приклад використання конструкції *switch-case*:

```
int monthNumber = 3;
String month;
switch (monthNumber) {
   case 1:  month = "Січень"; break;
   case 2:  month = "Лютий"; break;
   case 3:  month = "Березень"; break;
   case 4:  month = "Квітень"; break;
   case 5:  month = "Травень"; break;
```

```
    case 6:  month = "Червень"; break;
    case 7:  month = "Липень"; break;
    case 8:  month = "Серпень"; break;
    case 9:  month = "Вересень"; break;
    case 10: month = "Жовтень"; break;
    case 11: month = "Листопад"; break;
    case 12: month = "Грудень"; break;
    default: month = "Такий місяць не існує"; break;
}
```

Як сказано вище, у кожному блоці *case* є оператор *break*, який перериває свій блок коду. Його потрібно використовувати для переривання, інакше виконання коду продовжиться до першого зустріченого оператора *break* або до кінця конструкції *switch-case*, якщо не буде знайдено жодного оператора *break*. Іноді це зручно використовувати, наприклад:

```
int month = 3;
int year = 2020;
int numDays;
switch (month) {
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:    numDays = 31; break;
case 4:
case 6:
case 9:
```

```
case 11:    numDays = 30; break;
case 2:
   if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))
      numDays = 29;
   else
      numDays = 28;
   break;
default:
   System.out.print("Такий місяць не існує");
   break;
}
```

Допускається також використовувати вкладені оператори *switch*, але на практиці це не використовується.

Отже, підсумовуючи все вищесказане, можна виділити наступні правила, які застосовуються для оператора *switch*:

- змінні, які використовуються в операторі *switch*, можуть бути тільки цілі числа, класи-обгортки, рядки і перерахування;
- може бути будь-яка кількість операторів *case* в рамках одного *switch*. За кожним *case* слідує значення для порівняння, а потім йде двокрапка;
- значення *case* (константа або літерал) повинне бути того ж типу даних як і змінна в *switch*;
- коли змінна *switch* дорівнює виразу в операторі *case*, інструкції, які йдуть за *case*, будуть виконуватися до тих пір, поки не буде досягнутий оператор *break*;
- при досягненні оператора *break*, *switch* завершується, і керування переходить до наступного рядка після оператора *switch*;

- не кожен *case* може містити оператор *break*. Якщо відсутній оператор *break*, керування перейде до наступного оператору *case*. Так буде відбуватися до тих пір, доки не буде досягнутий *break*;
- оператор *switch* може мати додатковий оператор *default*, який має бути останнім в конструкції *switch-case*. Оператор *default* виконується у тому випадку, коли жоден з операторів *case* не виконується. Оператор *break* не є обов'язковим при описі оператора *default*.

Варто запам'ятати важливі властивості оператора *switch*:

- оператор *switch* відрізняється від оператора *if* тим, що може виконувати перевірку тільки рівності з обмеженою кількістю типів даних, а оператор *if* може обчислювати будь-які дії, що призводять до результату типу **boolean** або **Boolean**;
- дві константи *case* в одному і тому ж операторі *switch* не можуть мати однакові значення для порівняння;
- оператор *switch* ефективніше набору вкладених операторів *if*.

Порядок виконання роботи.

Основні завдання, які необхідно вирішити: навчитися працювати з конструкціями розгалуження, використати кожну конструкцію на практиці при виконанні завдання для самостійної роботи, порівняти між собою результати використання різних конструкцій розгалуження.

Розв'язання: Для рішення перелічених завдань необхідно скористатися програмним пакетом IntelliJ IDEA.

Приклади роботи з конструкціями розгалуження наведено вище.

Звіт з практикуму має містити: стислі теоретичні відомості, умову завдання, код програми з необхідним описом, результати виконання, висновки по роботі.

Завдання для виконання.

Частина 1. Для завдань з практичної роботи №3 (окрім практичної роботи №1) коректно реалізувати використання кожної з конструкцій розгалуження (*if-else, switch-case*, тернарного оператора) мінімум двічі в кожній програмі.

Частина 2. Використовуючи знання з практичної роботи №1 та практичної роботи №2, вивести в консоль таблицю істинності для трьох операнд: A, B та C (рис. 4.3).

A	B	C	A&B&C	A&B|C	A|B&C	A|B|C
false	false	false				
false	false	true				
false	true	false				
true	false	false				
true	true	false				
true	false	true				
false	true	true				
true	true	true				

Рисунок 4.3 – Таблиця істинності

Ускладнене завдання (за вибором). Для ускладненого завдання з практичної роботи №3 коректно реалізувати використання кожної з конструкцій розгалуження (*if-else, switch-case*, тернарний оператор) мінімум

двічі в кожній програмі таким чином, щоб конструкція *if-else* використовувалася тільки для одного гравця, а *switch-case* – тільки для іншого. Тернарний оператор використати для обох гравців.

Контрольні запитання.

1. Що таке розгалуження?
2. Для чого використовується розгалуження?
3. Які існують конструкції розгалуження?
4. Які існують логічні оператори?
5. Які існують оператори порівняння?
6. Яка різниця між конструкціями if-else та switch-case?
7. Які існують побітові оператори?
8. Які пріоритети логічних та побітових операторів?

Практична робота №5

Робота з циклами

Мета роботи: розглянути та порівняти можливості основних конструкцій циклів.

Теоретичні відомості

Цикли

Оголошення циклу дозволяє виконати одну і ту ж інструкцію або групу інструкцій декілька разів. На рис. 5.1 наводиться загальна структура оператора циклу в більшості мов програмування.

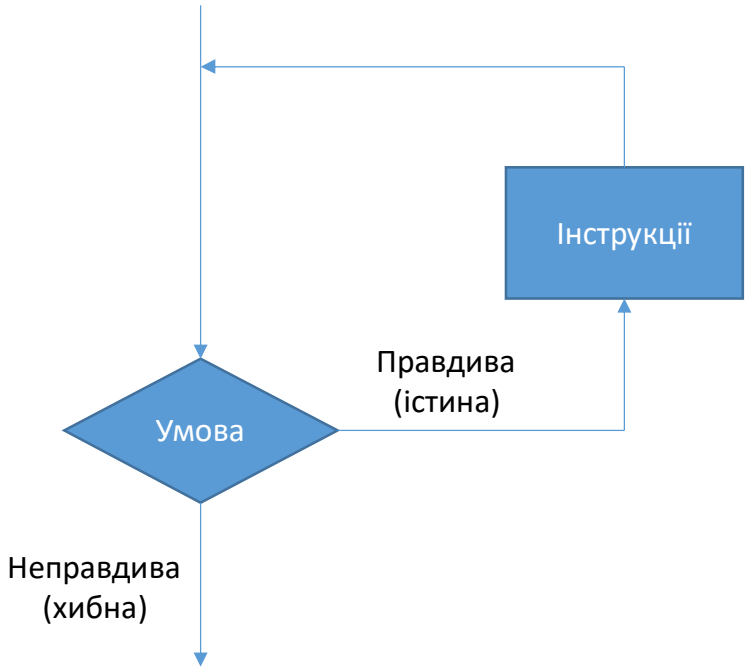

Рисунок 5.1 – Загальна схема циклу

Для забезпечення повторюваності обчислювального процесу в мові програмування Java введені оператори циклу (табл. 5.1). Послідовність операторів, які повинні виконуватися за один раз в операторі циклу, називається *ітерацією*.

Таблиця 5.1 – оператори циклу

for	Повторює інструкцію або групу інструкцій фіксовану кількість разів, яка визначається умовою.
while	Повторює інструкцію або групу інструкцій, поки задана умова є істинною (true). Цикл перевіряє умову до виконання тіла циклу.
do-while	Виконується так само, як і цикл while, за винятком того, що він перевіряє умову в кінці тіла циклу.

Цикл for

Цикл *for* (рис. 5.2) має структуру керування повтореннями, що дозволяє ефективно його використовувати, коли відомо скільки разів інструкції в тілі циклу повинні бути повторені.

Синтаксис циклу *for* в Java:

```
for (ініціалізація; логічний операнд; крок) {
  // Інструкції
}
```

Процес керування в циклі:

- спочатку виконується стадія ініціалізації (тільки один раз). Цей крок дозволяє оголошувати і форматувати змінні для керування циклом. Закінчується крапкою з комою ‘;’;
- далі йде логічний вираз (логічний операнд). Якщо він істинний, то тіло циклу виконується, якщо хибний – тіло циклу не буде виконано і контроль переходить до першої інструкції після циклу. Закінчується крапкою з комою ‘;’;
- після запуску тіла циклу на виконання, контроль переходить назад до кроку. Він дозволяє змінювати змінні для керування циклом. Записується без крапки з комою в кінці;
- після зміни кроку логічний вираз оцінюється знову. Якщо він істинний, то цикл виконується і процес повторюється, якщо хибний – цикл *for* завершується.

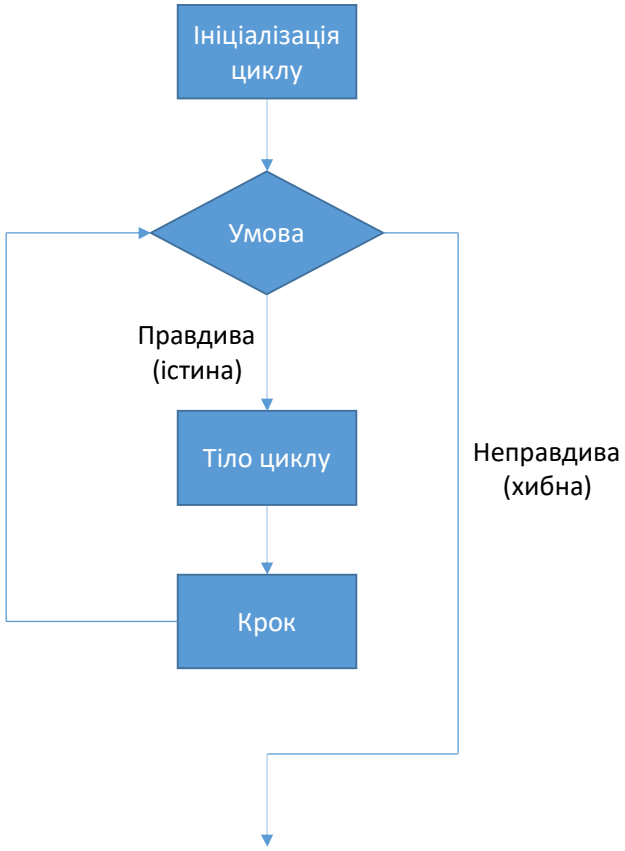

Рисунок 5.2 – Загальна схема циклу for

Приклад використання циклу *for*:

```
for(int x = 10; x < 15; x = x+1) {
   System.out.printf("Значення x: %d \n", x );
}
```

У результаті буде отримано:

Значення x: 10

Значення x: 11

Значення x: 12

Значення x: 13

Значення x: 14

У Java також присутній скорочений (його часто називають поліпшеним) цикл *for*, який здебільшого використовується для перебору елементів масиву або колекції. Синтаксис скороченого циклу *for*:

```
for (оголошення: вираз) {
   // Інструкції
}
```

де оголошення – створення локальної змінної (з тим же типом даних, що і у масиву), яка буде здійснювати доступ до елементів масиву; вираз – визначає масив, який необхідно перебрати в циклі. Вираз може бути заданий у вигляді статичного масиву, змінної або викликом методу, який повертає масив:

```
int [] array = {1, 2, 3, 4, 5};
   for(int i : array ){
      System.out.print( i + ", ");
   }
   System.out.print("\n");
   for( String name : {"Василь", "Дмитро", "Рустам", "Ганна"} ) {
      System.out.print( name + ", "); }
```

У результаті виконання даного коду буде отримано:

1, 2, 3, 4, 5,

Василь, Дмитро, Рустам, Ганна,

Цикл while

Цикл *while* багаторазово виконує тіло циклу до тих пір, поки умова є істиною, коли умова стає хибною – цикл завершується і програма передає керування першій інструкції, яка йде після циклу. Загальну схему циклу *while* зображено на рис. 5.3.

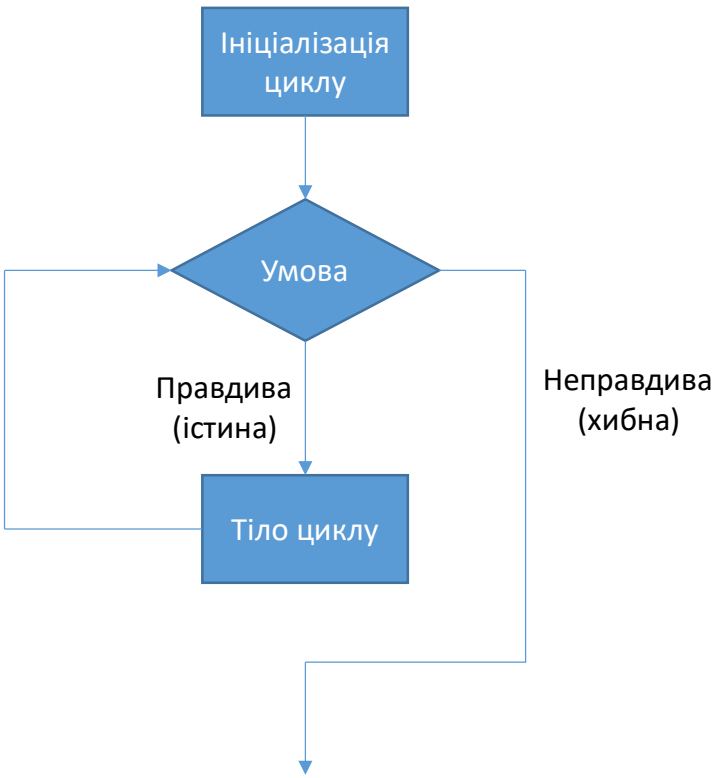

Рисунок 5.3 – Загальна схема циклу while

Синтаксис циклу while в Java:

while (логічний вираз) {

```
    // Інструкції
}
```

У вигляді умови може використовуватися будь-який логічний операнд або будь-яке ненульове значення.

Приклад використання циклу *while*:

```
int x = 1;
    while( x < 5 ) {
      System.out.printf("Значення x: %d \n", x );
      x++;
    }
```

У результаті в консоль буде виведено наступне:

Значення x: 1

Значення x: 2

Значення x: 3

Значення x: 4

Цикл do-while

Цикл *do-while* (рис. 5.4) схожий на цикл *while*, проте цикл *do-while*, на відміну від *while*, гарантовано виконається хоча б один раз.

Синтаксис циклу *do-while* в Java:

```
do {
   // Інструкції
} while (логічний вираз);
```

Якщо логічний вираз є істиною, то контроль переходить назад до тіла циклу, щоб виконати інструкції знову. Цей процес повторюватиметься до тих пір, поки логічний вираз не стане хибним.

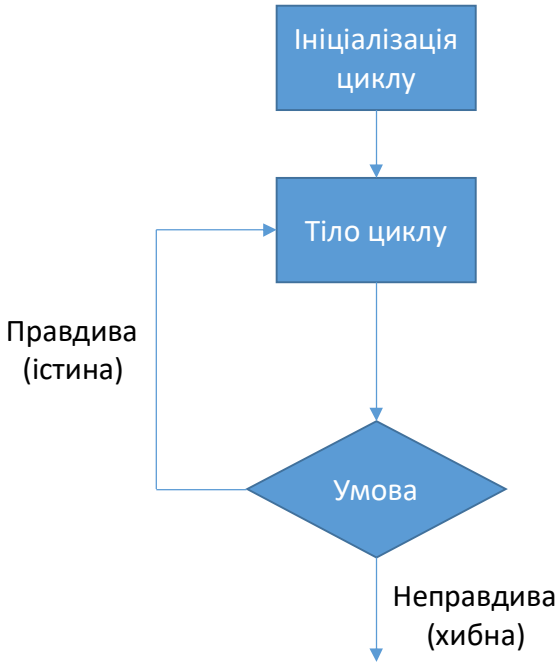

Рисунок 5.4 – Загальна схема циклу do-while

Приклад використання циклу *do-while*:

```
int x = 1;
    do {
      System.out.printf("Значення x: %d \n", x );
      x++;
    } while( x < 5 );
```

У результаті в консоль буде виведено наступне:

Значення x: 1

Значення x: 2

Значення x: 3

Значення x: 4

Найчастіше цикл *do-while* використовується для перевірки введення даних з клавіатури. Іноді (наприклад, для перевірки відповідності введеного типу даних) для рішення подібних задач може також застосовуватися цикл *while*.

Ключові слова break та continue

У мові програмування Java ключове слово *break* має наступні два використання:

- якщо ключове слово *break* зустрічається всередині циклу (рис. 5.5), то цикл переривається, а керування програмою переходить до першої інструкції після циклу;
- воно може бути використано для припинення роботи *case* в конструкції *switch-case* (приклад наведено в практичній роботі №4).

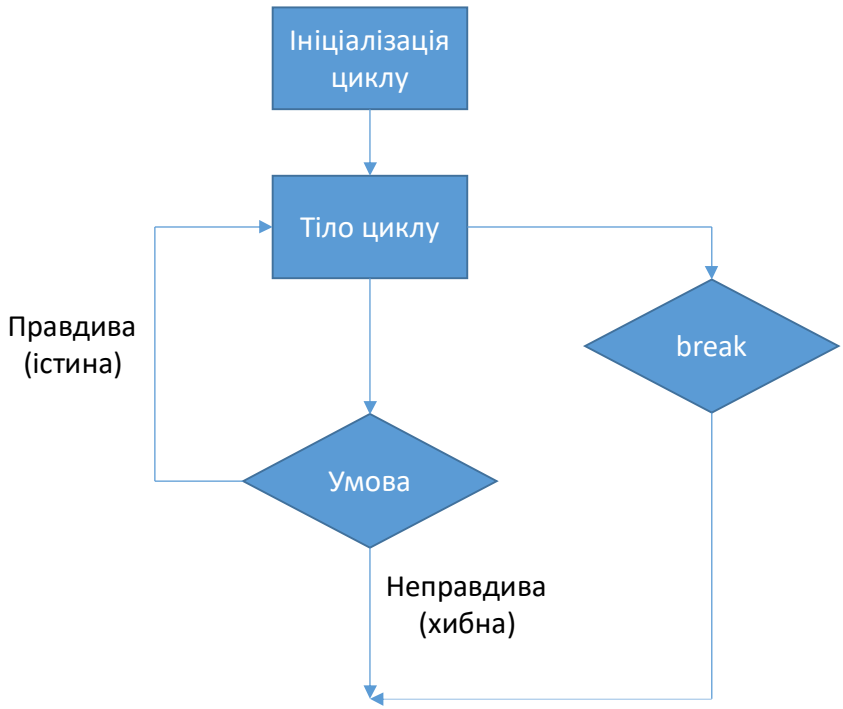

Рисунок 5.5 – Загальна схема циклу do-while з ключовим словом break

Наприклад,

```
int x = 1;
    do {
```

```
        System.out.printf("Значення x: %d \n", x );
        x++;
      if (x==3) break;
    } while( x < 5 );
```

призведе до наступного результату:

Значення x: 1

Значення x: 2

Ключове слово *continue* (рис. 5.6) може бути використане в будь-якій частині циклу. Його виконання призведе до переходу циклу на наступну ітерацію:

- у циклі *for* ключове слово *continue* викликає негайний перехід до кроку;
- у циклах *while* та *do-while* керування програмою відразу ж переходить у логічний вираз (умову).

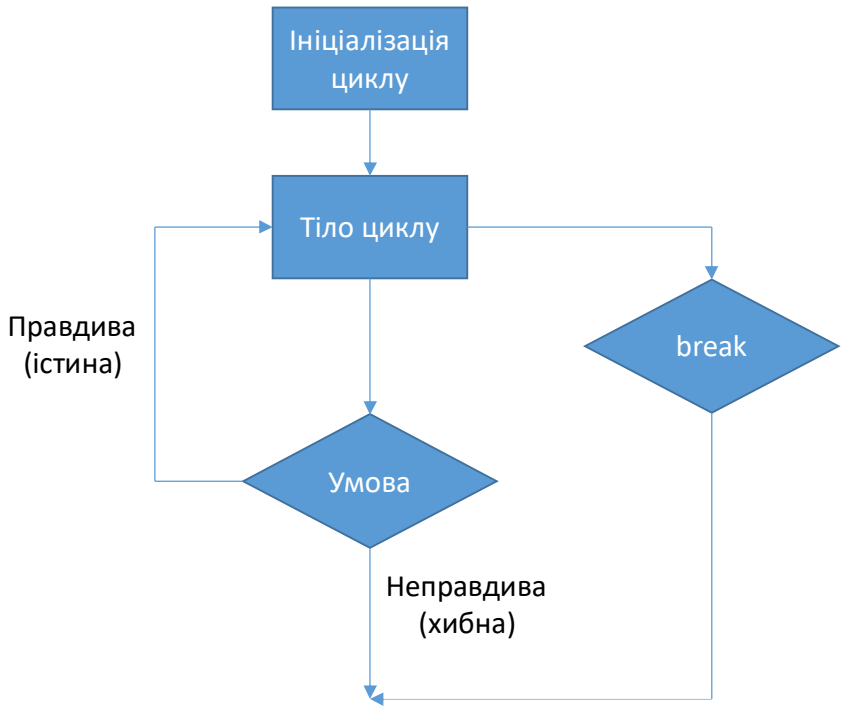

Рисунок 5.6 – Загальна схема циклу do-while з ключовим словом continue

Наприклад,

```
int x = 1;
    do {
      System.out.printf("Значення x: %d \n", x );
     if (x>=3) continue;
      x++;
    } while( x < 5 );
```

призведе до наступного результату:

Значення x: 1

Значення x: 2

Значення x: 3

Значення x: 3

...

Значення x: 3

У даному прикладі використання ключового слова *continue* перед операцією інкрементування призводить до виконання циклу, який ніколи не закінчиться.

Порядок виконання роботи.

Основні завдання, які необхідно вирішити: навчитися працювати з конструкціями циклів, використати кожен цикл на практиці при виконанні завдання для самостійної роботи, порівняти між собою результати використання різних циклів.

Розв'язання: Для рішення перелічених завдань необхідно скористатися програмним пакетом IntelliJ IDEA.

Приклади роботи з циклами наведено вище.

Звіт з практикуму має містити: стислі теоретичні відомості, умову завдання, код програми з необхідним описом, блок-схеми використаних циклів, результати виконання, висновки по роботі.

Завдання для виконання.

Частина 1. Для завдань з практичної роботи №4 за допомогою циклів коректно реалізувати використання перевірки введених з клавіатури даних (окрім типу даних String) усюди, де використовується **Scanner** або **BufferedReader**. Також у кожній з програм використати цикли **for** та **while** хоча б по одному разу.

Частина 2. Використовуючи цикли та знання з попередніх практичних робіт, реалізувати програму, в якій в консоль будуть виводитися квадрати парних чисел, починаючи зі введеного користувачем. У разі введення непарного числа почати виведення в консоль з першого (більшого за введене) парного числа.

Ускладнене завдання (за вибором). Для ускладненого завдання з практичної роботи №4 за допомогою циклів коректно реалізувати використання перевірки введених з клавіатури даних (окрім типу даних String) усюди, де використовується **Scanner** або **BufferedReader**, а також використати цикли **for** та **while** хоча б по одному разу.

Контрольні запитання.

1. Що таке цикл?
2. Які є цикли?
3. Чим відрізняються між собою різні конструкції циклів?
4. За що відповідають ключові слова break та continue?
5. Чи можна створити нескінченний цикл? Якщо так, то яким чином?

Практична робота №6

Створення, заповнення, сортування масивів

Мета роботи: розглянути та порівняти можливості масивів та колекцій.

Теоретичні відомості

Масиви

Масиви в Java – це структури даних фіксованого розміру, які зберігають елементи конкретного типу даних.

Одновимірні масиви в Java є списками однотипних змінних. Щоб створити масив, потрібно спочатку оголосити змінну масиву необхідного типу. Загальна форма оголошення одновимірного масиву виглядає наступним чином:

тип_даних[] ім'я_змінної;

де параметр '*тип_даних*' позначає тип даних елементів масиву; *ім'я_змінної* – змінна, яка містить посилання на масив у пам'яті.

Квадратні дужки можна ставити перед змінною (**float[]** salary;) або після неї (**int** price[];). Другий варіант є С-подібним і введений в Java для зручності переходу програмістів з C/C++ до Java. Незважаючи на це, більш правильним варіантом вважається саме перший (**float[]** salary;) – таким чином тип даних і квадратні дужки знаходяться в одному місці, що дозволяє з першого погляду зрозуміти, що оголошується масив певного типу:

Коли масив оголошений, пам’ять під нього ще не виділена. Для виділення пам’яті під масив, використовується ключове слово *new*, після якого знову вказується тип масиву і в квадратних дужках його розмір:

ім’я_змінної = new тип_даних[розмір_масиву];

Масив може бути оголошений та ініціалізований одним рядком:

int[] numbers = new int [13];

Після виконання цієї інструкції буде створено масив з 13 елементів. Кожному елементу присвоюється значення за замовчуванням для заданого типу (0 для типу даних **int**).

Для звернення до окремого елементу масиву або для зміни чи отримання його (елементу) значення після імені масиву в квадратних дужках необхідно задати індекс необхідного елемента:

numbers[5] = 17390;

numbers[9] = 15400;

Варто пам’ятати, що індекси масиву починаються з 0.

Окрім поелементної ініціалізації, ініціалізація масиву (відразу всього) може відбуватися за допомогою використання блоку для ініціалізації у випадках, коли заздалегідь відомі значення для кожного елемента масиву. У такому випадку замість new **int[13]**, у фігурних дужках через кому перераховуються значення елементів масиву. Розмір масиву розраховується компілятором з кількості зазначених у такий спосіб елементів:

int[] numbers = {39, 128, 188, 793, 1, 15, 19, 7, 30, 46, 21, 21};

Існує і третя форма ініціалізації масиву – безіменний масив. Він може використовуватися в двох випадках:

1) Оголошення та ініціалізація масиву з певним розміром, який потім з якоїсь причини повинен бути змінений, наприклад, масив має містити менше елементів. Повторно використовувати другу форму (блок ініціалізації) для ініціалізації масиву не можна – виникне помилка при компіляції:

```
int[] numbers = {1, 2, 3, 4};
numbers = {5, 6, 7}; // помилка компіляції
```

Для вирішення цієї проблеми можна, наприклад, використовувати безіменний масив, який створить новий масив в пам’яті. Форма створення безіменного масиву є поєднанням перших двох форм:

```
int[] numbers = {1, 2, 3, 4};
numbers = new int[]{5, 6, 7}; // немає помилок при компіляції
```

2) Інший випадок використання – ситуація, коли зберігати масив в окрему змінну відразу не потрібно, наприклад, передача масиву в метод. У наступному прикладі метод print приймає на вхід масив типу int:

```
public static void main(String[] args) {
    print(new int[]{1, 2, 3, 4});
}
public static void print(int[] array) {
    for (int element : array) {
        System.out.print(element + " ");
    }
}
```

Копіювання масивів

Масиви можна копіювати, наприклад, наступним чином:

```
public static void main(String[] args) {
    int[] array1 = {1,2,3};
    int[] array2 = array1;
    array1[2]=5;
    array2[2]=100;
```

```
    for(int element:array1){
        System.out.print(element+ " ");
    }
}
```

У результаті виконання коду в консоль буде виведено наступне:

1 2 100

Це означає, що подібна форма копіювання в окрему змінну копіює не масив, а лише посилання на нього, тобто обидві змінні посилаються на один і той же масив у пам'яті. Щоб отримати шляхом копіювання два незалежних один від одного масиви, можна скористатися наступним кодом:

```
public static void main(String[] args) {
    int[] array1 = {1,2,3};
    int[] array2 = new int[array1.length];
    for(int i=0; i<array1.length; i++){
        array2[i]=array1[i];
    }
}
```

При такій формі копіювання значення кожного окремого елементу по черзі записується у новий масив з таким же індексом. У даному прикладі використовується стандартне для масивів поле *length* (доступне тільки для читання і його значення не може бути змінене), яке містить довжину масиву.

Проте існують й інші методи копіювання даних з одного масиву в інший. Наприклад, клас *java.util.Arrays* призначений для роботи з масивами та містить зручні методи для роботи з масивами, наприклад:

- copyOf() – копіювання значень масиву;
- copyOfRange() – копіювання частини значень масиву;

- toString() – зображує всі елементи у вигляді одного рядка (наприклад, *[33, 12, 98]*);
- sort() – швидке сортування масиву;
- binarySearch() – бінарний пошук елементів масиву;
- fill() – заповнює масив переданим значенням (зручно використовувати, якщо необхідно значення за замовчуванням для масиву);
- equals() – перевіряє на ідентичність масиви;
- deepEquals() – перевіряє на ідентичність масиви масивів;
- asList() – повертає масив як колекцію.

Отже, метод Arrays.copyOf(початковий_масив, нова_довжина) – повертає масив-копію нової довжини. Якщо нова довжина менше оригінальної, то масив усікається до цієї довжини, а якщо більше, то доповнюється нулями:

```
public static void main(String[] args) {
   int[] array1 = {1,2,3};
   int[] array2 = Arrays.copyOf (array1, array1.length);
}
```

Метод Arrays.copyOfRange(початковий_масив, початковий_індекс, кінцевий_індекс) – також повертає масив-копію нової довжини, при цьому копіюється частина оригінального масиву від початкового індексу до кінцевого -1 (мінус одиниця).

```
public static void main(String[] args) {
   int[] array1 = {1,2,3};
   int[] array2 = Arrays.copyOfRange(array1, 0, 1);
} // результат: array2= {0}
```

Ще одним з варіантів копіювання є використання методу System.arraycopy(), який здійснює копіювання всього масиву або його частини в інший масив:

```
public static void main(String[] args) {
   int[] array1 = {1,2,3};
   int[] array2 = {0,0,0};
   System.arraycopy(array1, 0, array2, 1,2);
} результат: array2= {0, 1, 2}
```

Багатовимірні масиви

Багатовимірні масиви являють собою масиви масивів.

При оголошенні змінної багатовимірного масиву для вказівки кожного додаткового індексу (розмірності) використовується окремий ряд квадратних дужок, наприклад:

```
int[][] array = new int[8][6]; // двовимірний масив
int[][][] array = new int[8][6][9]; // тривимірний масив
```

Для того, щоб заповнити багатовимірний масив значеннями треба скористатися вкладеними циклами. Наприклад, у випадку двовимірного масиву для перебору рядків використовується зовнішній цикл *for*, для перебору стовпчиків – внутрішній цикл *for*. Спочатку відбувається виконання усіх ітерацій внутрішнього циклу, після чого зовнішній цикл переходить на наступну ітерацію і так до завершення усіх ітерацій зовнішнього циклу.

Наступний приклад демонструє, яким чином можна записати значення в двомірний масив 5x5:

```
public static void main(String[] args) {
```

```
int[][] array = new int[5][5];
int i, j, k = 0;
for (i = 0; i < 5; i++) {
    for (j = 0; j < 5; j++) {
        array[i][j] = k++;
        System.out.print(array[i][j] + " ");
    }
    System.out.println();
  }
}
```

У результаті виконання даного коду в консоль буде виведено наступний результат:

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19

20 21 22 23 24

Колекції

Колекції – це сховища, що підтримують різні способи накопичення і впорядкування об’єктів з метою забезпечення можливостей ефективного доступу до них.

Collection framework в мові Java складається з 3-х частин:

- інтерфейси – це абстрактні типи даних, які представляють колекції. Інтерфейси дозволяють управляти колекціями незалеж-

но від деталей їх подання. В об’єктно-орієнтованих мовах інтерфейси зазвичай формують ієрархію;

- класи – це конкретні реалізації інтерфейсів колекцій. По суті, вони є структурами даних, які можна багаторазово використовувати;
- алгоритми – це методи, які виконують необхідні обчислення з об’єктами, які реалізують інтерфейси колекцій, наприклад пошук і сортування. Алгоритми вважаються поліморфними, тобто один і той же метод може бути використаний в безлічі різних реалізацій відповідного інтерфейсу колекцій.

Вершиною ієрархії колекцій є інтерфейс *Collection*. Він визначає найменший набір методів, реалізованих усіма колекціями (табл. 6.1).

Таблиця 6.1 – Основні методи інтерфейсу Collection

№	Метод	Опис
	add(E obj)	Додає об’єкт до колекції. Повертає true, якщо об’єкт був доданий до колекції.
	clear()	Видаляє всі елементи колекції.
	contains(Object obj)	Повертає true, якщо об’єкт є елементом колекції. В іншому випадку повертає false.
	containsAll(Collection<> c)	Повертає true, якщо колекція містить всі елементи с. В іншому випадку повертає false.
	equals(Object obj)	Повертає true, якщо колекція і об’єкт еквівалентні. В іншому випадку повертає false.
	hashCode()	Повертає хешкод колекції.

Продовження таблиці 6.1

	isEmpty()	Повертає true, якщо колекція порожня. В іншому випадку повертає false.
	iterator()	Повертає ітератор колекції.
	remove(Object obj)	Видаляє один екземпляр об'єкта з колекції. Повертає true, якщо елемент був видалений. В іншому випадку повертає false.
	removeAll(Collection<> c)	Видаляє всі елементи з колекції. Повертає true, якщо в результаті колекція змінюється (тобто елементи вилучені). В іншому випадку повертає false.
	retainAll(Collection <> c)	Видаляє з колекції всі елементи крім зазначених. Повертає true, якщо в результаті колекція змінюється (тобто елементи вилучені). В іншому випадку повертає false.
	size()	Повертає кількість елементів, що містяться в колекції.
	toArray()	Повертає масив, що містить всі елементи колекції. Елементи масиву є копіями елементів колекції.

Класи ArrayList та LinkedList

Java надає набір стандартних класів колекцій, які реалізують інтерфейси *Collection*. Деякі з класів надають повні реалізації, які можуть бути використані як є, а інші є абстрактними класами, надаючи скелетні реалі-

зації, які використовуються як початкові точки для створення конкретних колекцій. **ArrayList** та **LinkedList** є одними зі стандартних класів колекцій в Java.

ArrayList – це масив, який автоматично розширюється. ArrayList – найпопулярніший вид колекцій (списків).

Масиви мають фіксовану довжину протягом свого життєвого циклу. Це значить, що після створення масиву його довжина більше не може бути змінена. На відміну від класичних масивів, ArrayList може змінювати свій розмір під час виконання програми, при цьому не обов'язково вказувати розмір при створенні об’єкта. Ще однією перевагою ArrayList є можливість додавання (або видалення) нового елементу в середину (або будь-яке інше місце) колекції. Елементи ArrayList можуть мати будь-який тип даних, а також значення *null*. Це зручно, коли заздалегідь не відомо точного розміру масиву.

Працювати з ArrayList просто:

1) Створюється потрібний об'єкт, наприклад, *ArrayList studentNames = new ArrayList();*
2) Додати новий об’єкт в колекцію можна за допомогою методу add(). Для звернення до необхідного елементу колекції використовується метод get(). Індексування елементів використовується так само, як і для звичайних масивів, але без квадратних дужок. ArrayList також містить метод size(), який повертає поточну кількість елементів в масиві (у звичайному масиві використовується властивість length).

Імена об’єктам колекцій прийнято давати у множині, наприклад, studentNames, songs і так далі.

Переваги класу **ArrayList**:

- швидкий доступ до елементу колекції за індексом;

- швидка вставка і видалення елементів з кінця.

Серед недоліків класу **ArrayList** можна відзначити повільну вставку і видалення елементів у будь-яке місце колекції окрім кінця.

Пов'язаний масив LinkedList реалізує інтерфейси List, Dequeue, Queue і є представником двонаправленого списку, де кожен елемент структури містить посилання на попередній і наступний елементи. Ітератор підтримує обхід колекції в обидві сторони. Як і ArrayList, LinkedList реалізує методи отримання, видалення і вставки елементів у початок, середину і кінець списку, а також дозволяє додавати будь-які елементи, у тому числі і null.

Відмінність LinkedList від ArrayList полягає в тому, що LinkedList виконує вставку і видалення елементів у списку за фіксований час, який є меншим за час, необхідний для тієї ж операції в ArrayList. У більшості випадків LinkedList програє ArrayList і по споживаній пам'яті, і по швидкості виконання операцій. Але якщо в алгоритмі передбачена активна робота (вставка/видалення) у середині списку або у випадках, коли необхідно гарантований час додавання елемента в список, то доцільно використовувати LinkedList.

Порядок виконання роботи.

Основні завдання, які необхідно вирішити: навчитися працювати з одновимірними та багатовимірними масивами, а також колекціями, використати кожну колекцію на практиці при виконанні завдання для самостійної роботи, порівняти між собою результати використання масивів та колекцій.

Розв'язання: Для рішення перелічених завдань необхідно скористатися програмним пакетом IntelliJ IDEA.

Приклади роботи з масивами наведено вище.

Приклади використання колекцій ArrayList та LinkedList:

```
import java.util.ArrayList;
import java.util.List;
public class ArrayListTest {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        System.out.println("Початковий розмір колекції: " + list.size());
        list.add("S");
        list.add("A");
        list.add("G");
        list.add("H");
        list.add("H");
        list.add("F");
        list.add("O");
        list.add(1, "S");
        list.set(0, "A");
System.out.println("Розмір колекції після додавання: "+ list.size());
        list.remove("H");
System.out.println("Розмір колекції після видалення: "+ list.size());
        System.out.println("Вміст колекції: " + list);
        System.out.println(list.get(0));
    }
}
```

```
import java.util.LinkedList;
public class LinkedListTest {
    public static void main(String[] args) {
```

```
        LinkedList<String> countries = new LinkedList<String>();
        countries.add ("Латвія");
        countries.add ("Франція");
        countries.addLast ("Україна");
        countries.addFirst("Іспанія");
        countries.add (1, "Італія");
    System.out.printf("Колекція містить %d елементів \n", countries.size());
        System.out.println(countries.get(1));
        countries.set(1, "Данія");
        for (String state: countries){
           System.out.println(state);}
        if (countries.contains("Польща")){
           System.out.println("Колекція містить країну Польща");}
        countries.remove("Польща");
        countries.removeFirst();
        countries.removeLast();
      }
    }
```

Звіт з практикуму має містити: стислі теоретичні відомості, умову завдання, код програми з необхідним описом, результати виконання, висновки по роботі.

Завдання для виконання.

Частина 1. Створити одновимірний масив цілих чисел, довжину та значення якого вводить користувач з клавіатури. Визначити значення, які

повторюються, та їх кількість (для кожного окремо). Організувати перевірку коректності введених даних.

Виконати це ж завдання, використовуючи колекції.

Частина 2. Створити одновимірний масив цілих чисел, довжину та значення якого вводить користувач з клавіатури. Організувати перевірку коректності введених даних. Змістити усі значення даного масиву на значення, яке вводить користувач з клавіатури. При цьому останнє значення масиву має зміщатися в перший елемент. Кожну ітерацію вивести на екран.

Наприклад, є масив [1, 7, 3, 5, 8, 2, 4] і користувач вводить значення 5. У результаті на екран буде виведено:

Уведіть початковий масив:

1, 7, 3, 5, 8, 2, 4

Початковий масив: [1, 7, 3, 5, 8, 2, 4]

На скільки значень змістити його елементи?

5

Відбувається зміщення елементів масиву:

[4, 1, 7, 3, 5, 8, 2]

[2, 4, 1, 7, 3, 5, 8]

[8, 2, 4, 1, 7, 3, 5]

[5, 8, 2, 4, 1, 7, 3]

Результуючий масив: [3, 5, 8, 2, 4, 1, 7].

Уведення початкового масиву можна організувати іншим способом, наприклад, стовпчиком, підтверджуючи кожен з елементів натисканням клавіші ‘Enter’.

Ускладнене завдання (за вибором). Створити двовимірний масив цілих чисел, розмірність **MxN** якого вводить користувач з клавіатури. Організувати перевірку коректності введених даних. Змістити усі значення

даного масиву на значення, яке вводить користувач з клавіатури. При цьому останнє значення першого рядка має зміщатися в перший елемент другого рядка і так далі. Кожну ітерацію вивести на екран.

Наприклад, є масив

1, 7, 3

2, 5, 9

4, 8, 6

і користувач вводить значення 4. У результаті на екран буде виведено:

Уведіть початковий масив:

1, 7, 3

2, 5, 9

4, 8, 6

Початковий масив: 1, 7, 3

2, 5, 9

4, 8, 6

На скільки значень змістити його елементи?

4

Відбувається зміщення елементів масиву:

6, 1, 7

3, 2, 5

9, 4, 8

8, 6, 1

7, 3, 2

5, 9, 4

4, 8, 6

1, 7, 3

2, 5, 9

Результуючий масив: 9, 4, 8

6, 1, 7

3, 2, 5

Уведення початкового масиву можна організувати іншим способом, наприклад, стовпчиком, підтверджуючи кожен з елементів натисканням клавіші ‘Enter’.

Контрольні запитання.

1. Що таке колекція в Java?
2. Які є колекції?
3. Чим відрізняються між собою різні колекції?
4. Що таке масив? Які бувають масиви?
5. Які є варіанти оголошення масивів?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1 Gaddis T., Starting Out with Java: From Control Structures through Objects / T. Gaddis. – London: Pearson, 2018. – 1168 p.

2 Schildt H., Java: A Beginner's Guide / H. Schildt. – NY: McGraw-Hill Education, 2017. – 752 p.

3 Sierra K. and Bates B., Head First Java / K. Sierra, B. Bates. – USA: O'Reilly Media, 2005. – 688 p.

4 Payne B., Learn Java easy way / B. Payne. – San Francisco: No Starch Press, 2018. – 292 p.

5 McAllister W. and Fritz S. Jane, Programming Essentials Using Java: A Game Application Approach / W. McAllister and S. Jane Fritz. – Boston: Mercury Learning & Information, 2017. – 540 p.

6 Bloch J., Effective Java / J. Bloch. – USA: Addison-Wesley Professional, 2018. – 412 p.